

Vircon32: Cómo crear juegos

Documento con fecha 2021.05.19

Escrito por Carra

¿Qué es esto?

Este documento es una guía rápida para empezar a crear juegos para la consola virtual Vircon32. El objetivo es comenzar desde cero y dar una serie de pasos a seguir para saber cómo crear juegos muy básicos.

Índice

Esta guía está organizada en pequeñas secciones, que nos explican primero el proceso de forma global para después enfocarse en cada uno de los aspectos que necesita un juego: imagen, sonido, control del jugador, etc.

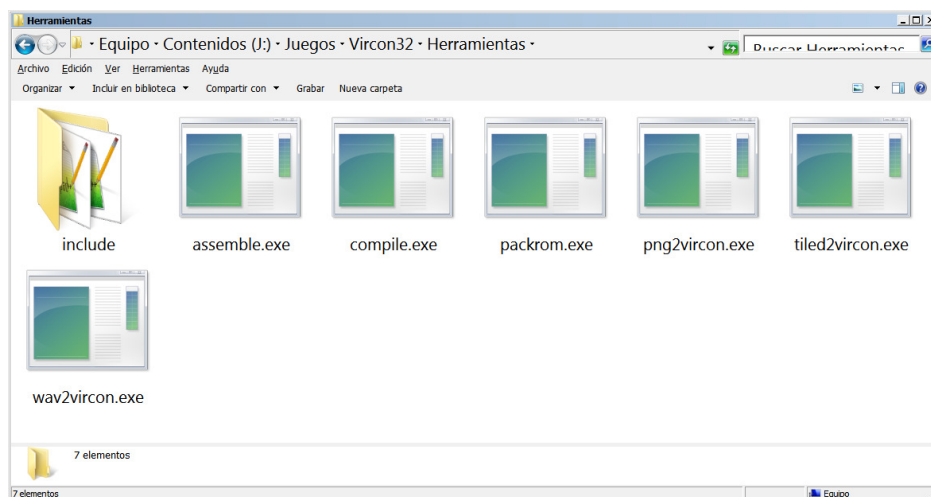
Introducción	2
Las herramientas de desarrollo	2
Empaquetar una ROM.....	4
Automatizar el proceso	5
Estructura típica de un juego	6
Interpretar errores del compilador.....	8
Leer el estado de los mandos	9
Dibujar personajes y objetos.....	9
Escenarios con scroll.....	11
Efectos de sonido	11
Música de fondo	12
Manejar objetos del juego	12
Dónde seguir aprendiendo.....	13

Introducción

Esta guía se centra sobre todo en el propio proceso de escribir un programa en C y usar las herramientas de desarrollo para poder ejecutarlo en la consola. Daremos también ejemplos típicos de cómo usar algunas funciones de la consola, pero en esta guía no entramos al detalle de todo lo que puede hacerse en Vircon32. Para comprender mejor los sistemas de la consola es recomendable leer la guía sobre su funcionamiento.

Las herramientas de desarrollo

Cuando descargamos las herramientas de desarrollo de Vircon32, encontraremos una carpeta con el siguiente contenido:



Todos estos ejecutables son programas de línea de comandos que podemos automatizar ya sea escribiendo scripts o usando estas herramientas desde otros programas como por ejemplo un entorno de desarrollo (IDE). Para hacer esto normalmente necesitaremos añadir la carpeta DevTools a las rutas de ejecutables de Windows, usando PATH.

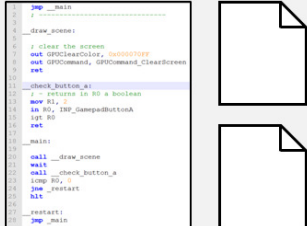
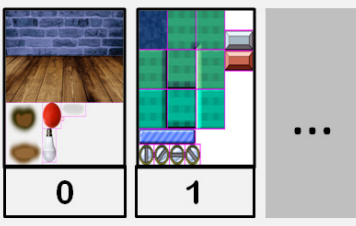
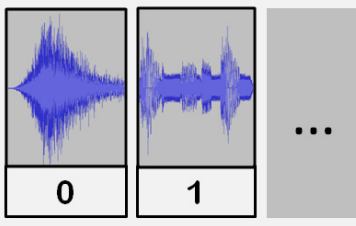
Vamos a ver qué son cada uno de estos elementos:

- **compile.exe** es el compilador de C. Traduce nuestros programas en lenguaje C a código ensamblador.
- **assemble.exe** es el ensamblador. Convierte el código ensamblador a instrucciones en formato binario que la CPU de Vircon32 puede entender.
- **packrom.exe** es el empaquetador de roms. Toma el programa binario y le añade las imágenes y sonidos que necesita usar. El resultado es un único archivo .v32 que el emulador de Vircon ya puede cargar.
- **png2vircon.exe** convierte imágenes en formato PNG al formato interno que puede usar el chip gráfico de Vircon32.
- **wav2vircon.exe** convierte sonidos en formato WAV al formato interno que puede usar el chip de sonido de Vircon32.

- **tiled2vircon.exe** es una herramienta que importa mapas de tiles creados en el programa Tiled. Convierte cada capa en un archivo binario que podemos incluir en el cartucho para usarlo como un array 2D.
- **include:** esta carpeta contiene los archivos de la librería estándar de C para el compilador.

Los pasos para crear un programa

La consola Vircon32 funciona con cartuchos virtuales, que al igual que en otros emuladores son archivos de roms. Cada juego es un único archivo en el que se empaqueta todo lo que el juego necesita. Ese contenido se divide en 3 partes independientes:

ROM de Programa	ROM de Video	ROM de Audio
<p>Instrucciones + Datos</p> 	<p>Lista de imágenes</p> 	<p>Lista de sonidos</p> 

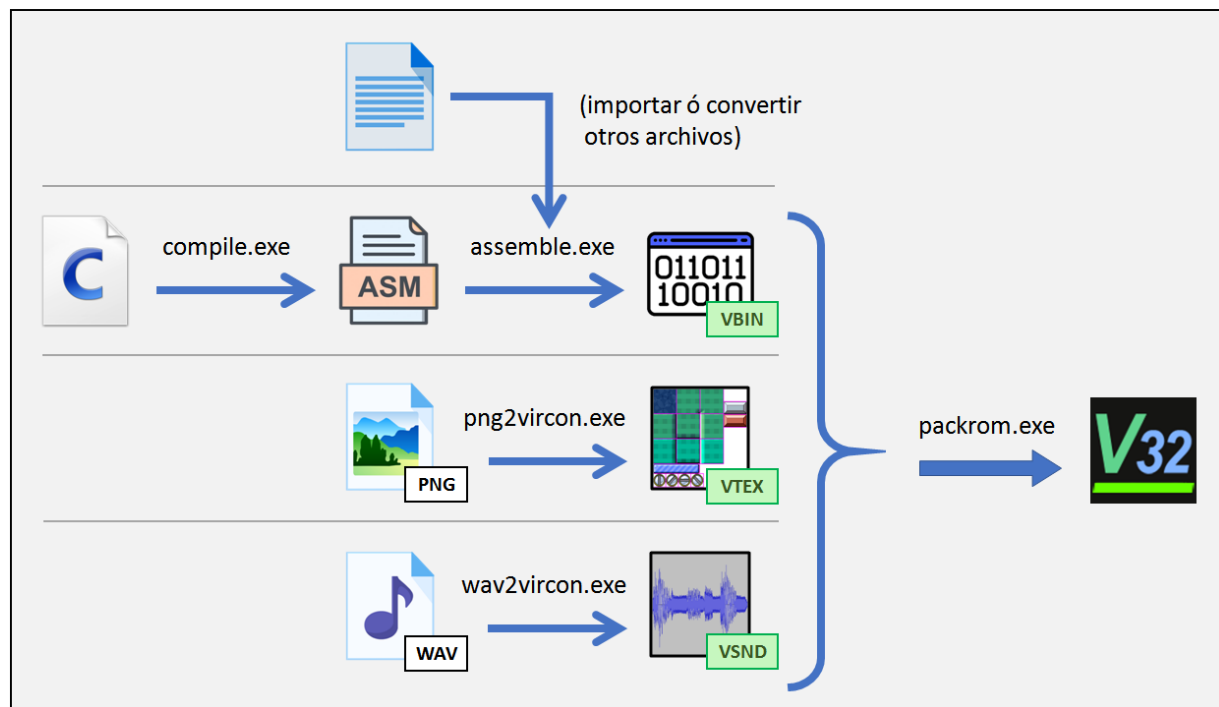
Si los cartuchos de Vircon32 sólo incluyeran el programa, crear un juego de Vircon32 a partir del código sólo supondría estos 2 pasos:



Sin embargo, en Vircon32 con esto sólo habríamos creado la rom de programa. También vamos a necesitar incluir en el cartucho una serie de imágenes y sonidos para que el programa los pueda usar. En general los pasos a seguir son los siguientes:

- 1) Compilar y ensamblar nuestro programa en C
- 2) Convertir todas las imágenes al formato nativo de Vircon32
- 3) Convertir todos los sonidos al formato nativo de Vircon32
- 4) Usar el empaquetador para unir todo en el archivo de la rom final
- 5) Usar el emulador para probar nuestro programa

Es posible que también tengamos algunos archivos con datos adicionales (como mapas de tiles) que necesitemos importar. En general el proceso sigue este esquema:



Los archivos que se marcan en color verde son los que ya están en el formato nativo de Vircon32, y por tanto ya se pueden incluir en el empaquetado final.

Empaquetar una ROM

El empaquetador puede necesitar incluir muchos archivos en la rom (un juego de Vircon32 puede tener hasta 256 imágenes y 1024 sonidos). Por eso, darle los archivos por línea de comandos no sería muy práctico.

En su lugar usa como parámetro de entrada un archivo XML que contiene la definición completa de la rom, incluyendo también algunos detalles generales como título y versión. En los ejemplos se puede ver cómo se usa, pero la forma del documento XML es esta:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<rom-definition version="1.0">
  <rom type="cartridge" title="Game title" version="1.0" />
  <binary path="PathToBinary.vbin" />
  <textures>
    <texture path="PathToTexture1.vtex" />
    <texture path="PathToTexture2.vtex" />
  </textures>
  <sounds>
    <sound path="PathToSound1.vsnd" />
    <sound path="PathToSound2.vsnd" />
  </sounds>
</rom-definition>
```

El orden en el que aparecen las texturas y sonidos es el mismo en el que se empaquetarán en el archivo. Es decir, nuestro programa podrá acceder a la primera textura de la lista con la ID 0, a la segunda con la ID 1, etc.

Un programa puede no tener texturas ni sonidos: el binario del programa es la única de las 3 partes que es obligatoria. Sin embargo el XML siempre debe tener los elementos `<textures>` y `<sounds>` aunque estén vacíos.

Automatizar el proceso

El empaquetador sólo hace el último paso para crear nuestro juego, con el programa ya compilado y los sonidos e imágenes ya convertidos. Crear un juego supone crear nuestra rom de nuevo cada vez que hacemos cambios, así que lo normal es que queramos usar algún script o herramienta externa para automatizarlo con un solo click.

La manera más sencilla es escribir un pequeño script de línea de comandos para hacer esto. En los fuentes de programas incluidos se pueden ver ejemplos prácticos, pero vamos a ver aquí este ejemplo sencillo de un archivo de comandos BAT de Windows.

```
@echo off

echo Compile the C code
echo -----
compile Program.c -o obj\Program.asm || goto :failed

echo Assemble the ASM code
echo -----
assemble obj\ Program.asm -o obj\Program.vbin || goto :failed

echo Convert the PNG textures
echo -----
png2vircon Texture.png -o obj\Texture.vtex || goto :failed

echo Convert the WAV sounds
echo -----
wav2vircon Sound.wav -o obj\Sound.vsnd || goto :failed

echo Pack the ROM
echo -----
packrom RomDefinition.xml -o bin\FinalGame.v32 || goto :failed
goto :succeeded

:failed
echo BUILD FAILED
exit /b %errorlevel%

:succeeded
echo BUILD SUCCESSFUL
exit /b

@echo on
```

Este script se podría simplificar pero lo importante es que llama a todas las etapas y, si en alguna de ellas se produce un error, detiene el proceso en vez de intentar continuar.

Algo que también podemos ver en el script es que utiliza los subdirectorios obj y bin. Programando en C (no sólo en Vircon32) es habitual usar una estructura de carpetas como esta en cada proyecto. Así podemos separar el resultado final (en “bin”), los archivos intermedios que se van generando (en “obj”), y los propios archivos fuente del proyecto.

Estructura típica de un juego

Ya sabemos crear el juego a partir del programa en C. Ahora vamos a ver cómo escribir el programa en sí. Esto no pretende enseñar a programar a quien no lo ha hecho nunca, ni tampoco es una guía completa sobre cómo programar juegos. Pero sí vamos a dar algunas nociones básicas con ejemplos, y a ver qué prácticas se suele seguir al programar juegos.

En general un juego en Vircon32 se suele programar siguiendo esta estructura:

- Incluir librerías
- Dar nombre a nuestras texturas, regiones y sonidos
- Declarar variables globales
- Funciones auxiliares
- **Función main**
 - Configurar texturas y sonidos
 - Inicializar la partida
- **Bucle principal**
 - Leer los mandos y aplicar las acciones del jugador
 - Simular las mecánicas de nuestro juego
 - Dibujar en pantalla la escena y los objetos en ella
 - Esperar al siguiente frame para controlar la velocidad

El elemento más importante en esta estructura es el bucle principal. Normalmente los juegos siempre se repiten en un bucle infinito, y las acciones de este bucle se realizan una vez por cada frame. En Vircon32 esto supone que lo repetimos 60 veces por segundo.

En la página siguiente podemos ver un pequeño ejemplo completo que sigue esta estructura de programa. Este programa es muy simple (sólo dibuja al personaje sobre un fondo fijo y nos permite moverlo), pero nos permite ver en una sola página la estructura en la práctica. Además las diferentes partes están señaladas con comentarios.

```

// include Vircon libraries
#include "video.h"
#include "input.h"
#include "time.h"

// -----
// DEFINITIONS

// give names to our texture regions
#define RegionBackground 0
#define RegionRobotRight 1

// -----
// MAIN FUNCTION

void main( void )
{
    // -----
    // PART 1: CONFIGURE OUR TEXTURES

    select_texture( 0 );

    // define our texture regions
    select_region( RegionBackground );
    define_region_topleft( 0,0, 639,359 );
    select_region( RegionRobot );
    define_region( 1,361, 66,441, 33,417 );

    // -----
    // PART 2: INITIALIZATIONS

    // our robot starts at the screen center
    int RobotX = screen_width / 2;
    int RobotY = screen_height / 2;

    // -----
    // PART 3: MAIN LOOP

    // keep repeating our game logic for every frame (60 fps)
    while( true )
    {
        // move robot in the direction we press
        int DirectionX, DirectionY;
        gamepad_direction( &DirectionX, &DirectionY );
        RobotX += 2 * DirectionX;
        RobotY += 2 * DirectionY;

        // draw our background to fill the screen
        select_region( RegionBackground );
        draw_region_at( 0, 0 );

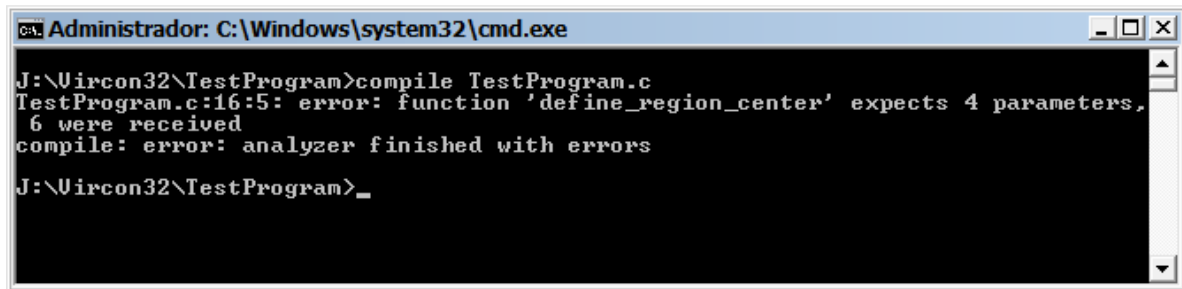
        // now draw robot in its current position
        select_region( RobotImage );
        draw_region_at( RobotX, RobotY );

        // wait for next frame to control game speed
        end_frame();
    }
}

```

Interpretar errores del compilador

Es normal que cometamos errores en nuestros programas, y nos interesa saber cómo encontrar un error cuando el compilador nos lo avisa. El compilador de C de Vircon32 da los errores en el mismo formato que el compilador gcc, es decir: nombre de archivo, línea y columna. Podéis verlo en este ejemplo:



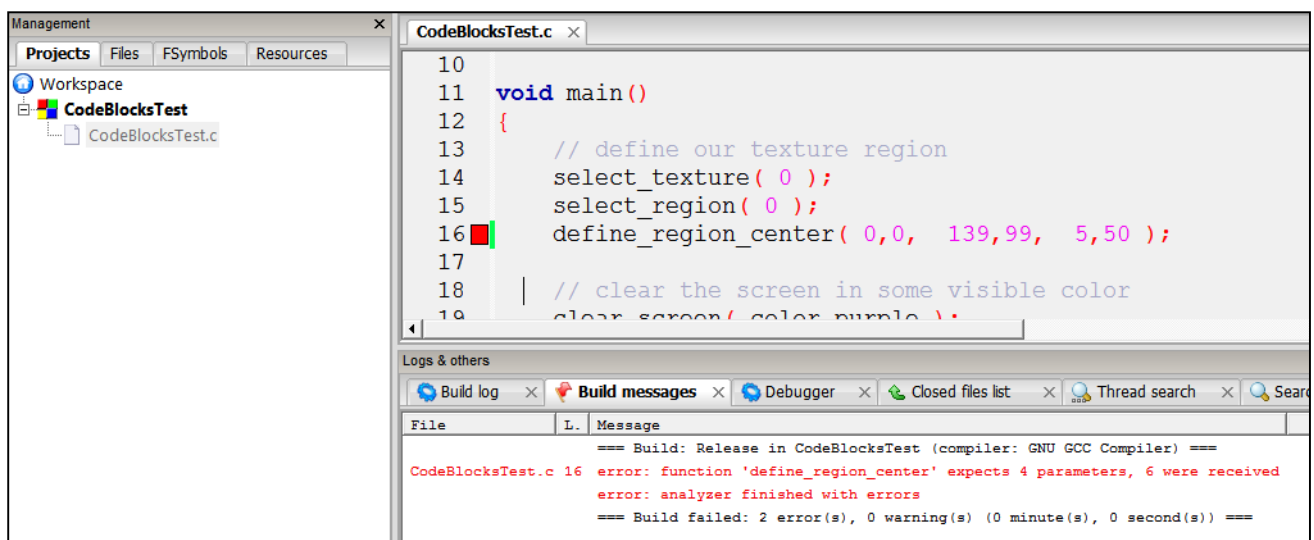
```
Administrador: C:\Windows\system32\cmd.exe

J:\Vircon32\TestProgram>compile TestProgram.c
TestProgram.c:16:5: error: function 'define_region_center' expects 4 parameters,
6 were received
compile: error: analyzer finished with errors

J:\Vircon32\TestProgram>_
```

Si programamos sin un IDE podemos buscar estos errores a mano con algún editor de texto que incluya números de línea, como por ejemplo Notepad++. Sin embargo, como el compilador se ha diseñado para comportarse como gcc (hasta cierto punto), es posible trabajar con un IDE configurándolo correctamente.

Entre los fuentes de ejemplo que podéis descargar, existe uno llamado “CodeBlocksTest”, que es precisamente un proyecto ya creado para Code::Blocks. Con él podemos no sólo compilar nuestro programa sino también usar el botón Run para probar la rom en el emulador. En esta imagen podéis ver cómo nos muestra Code::Blocks el mismo error.



Aunque no se ha probado este método en otros IDEs, es muy posible que se pueda trabajar con ellos también. Lo que se hace en este proyecto de ejemplo es configurar nuestros archivos C para que Code::Blocks **NO** los compile. En su lugar, decimos al IDE que ejecute nuestro archivo BAT (como el que mostramos anteriormente) como evento post-compilación.

Y con esto ya nos hacemos una idea de cómo trabajar con un programa a nivel global. Lo que haremos en las siguientes secciones es hablar de algunas acciones que normalmente se necesitan en un juego, y mostrar maneras en las que podemos llevar a cabo.

Leer el estado de los mandos

Lo primero que solemos necesitar hacer en el bucle principal es saber lo que el jugador está pulsando para actualizar el juego. Aquí mostramos 2 alternativas: comprobar direcciones individuales o comprobarla globalmente.

```
// aqui nos movemos solo en X, ignoramos las direcciones verticales
if( gamepad_left() > 0 ) Position.x -= Speed;
if( gamepad_right() > 0 ) Position.x += Speed;

// aqui nos movemos en cualquier direccion automaticamente
gamepad_direction( &DirectionX, &DirectionY );
Position.x += Speed * DirectionX;
Position.y += Speed * DirectionY;
```

¿Qué ocurre con los botones? En Vircon32 no hay eventos. Para saber si un botón ya estaba pulsado ó se acaba de pulsar ahora mismo, normalmente necesitaríamos guardar el estado del frame anterior para ver si ha cambiado. Sin embargo esto no es necesario: la consola ya nos da esta información. Podemos detectarlo así:

```
// leemos el estado del boton A
int ButtonA = gamepad_button_a();

// cuando pulsamos el boton A empezamos a cargar energia
if( ButtonA == 1 )
    StoredEnergy = 1;

// por cada frame que sigue pulsado, la energia aumenta
else if( ButtonA > 0 )
    StoredEnergy++;

// al soltar el boton, disparamos
if( ButtonA == -1 )
    Shoot();
```

Dibujar personajes y objetos

Es común que en un juego los personajes puedan tener animaciones. Un método sencillo para hacerlo es definir varias regiones consecutivas y según el número de frames ir haciendo un ciclo por todas ellas.

Supongamos por ejemplo que tenemos nuestro personaje dibujado así:



Las librerías de C nos permiten definir estas regiones consecutivas de una sola vez. En este ejemplo podemos ver cómo se hace esto, y cómo se animaría al dibujarlo:

```
// definimos las regiones de nuestra animacion
define_region_matrix
(
    100,    // ID de la primera region a crear
    1,1,    // punto inicial del primer cuadro
    50,50,  // punto final del primer cuadro
    25,47,  // punto de referencia del primer cuadro
    7,1,    // la matriz tiene 7 cuadros en X y 1 en Y
    1      // los cuadros tienen una separacion de 1 pixel
);

// cambiamos de cuadro cada 5 frames
AnimationTime++;

if( AnimationTime % 5 == 0 )
    AnimationImage++;

// al acabarse, la animacion se repite desde el principio
if( AnimationImage >= 7 )
    AnimationImage = 0;

// cada frame dibujamos nuestra animacion
select_region( 100 + AnimationImage );
draw_region_at( PlayerX, PlayerY );
```

En muchos juegos no se dibuja los personajes mirando a ambos lados, sino que siempre se dibuja hacia la derecha y para que mire a la izquierda se voltea la imagen. Podemos conseguir esto con un escalado en X igual a -1.

```
// dibujar mirando a la derecha
if( PlayerSpeedX >= 0 )
    draw_region_at( PlayerX, PlayerY );

// dibujar mirando a la izquierda
else
{
    set_drawing_scale( -1, 1 );
    draw_region_zoomed_at( PlayerX, PlayerY );
}
```

Escenarios con scroll

Las texturas que usa Vircon32 (1024x1024 pixels) son más grandes que la pantalla (640x360), pero aún así si queremos hacer escenarios grandes se nos quedarán pequeñas. La consola nos permite usar varias texturas y conectarlas, pero la manera más usada en muchos juegos es crear mapas de tiles.

Entre los programas que se incluyen en DevTools está [tiled2vircon.exe](#). Este programa nos permite importar mapas de tiles hechos con el editor Tiled. La manera de hacerlo es llamar al importador antes de compilar el programa en C para que el archivo importado ya esté creado. Si por ejemplo creamos un archivo llamado "Level1.map" en la carpeta obj del proyecto, desde C podemos importar esto en nuestro programa como si fuera un array de esta forma:

```
// el contenido del archivo se guarda como un array en el cartucho
embedded int[ TilesInY ][ TilesInX ] MapBricks = "obj\\Level1.map";
```

Luego, en nuestras texturas tendremos las imágenes de cada tile con las que dibujar el mapa. Para definir el conjunto de tiles en el programa usaremos al igual que antes la función [define_region_matrix\(\)](#).

Después, para dibujar el mapa en pantalla, podemos usar un bucle que recorra el mapa en X y en Y. ¡Pero cuidado! Un mapa puede ser muy grande, y tratar de dibujarlo entero cada frame puede exigir demasiado a la consola. En ese caso debemos comprobar qué rango de tiles quedan dentro de la pantalla y limitar el bucle a ese rango en X y en Y.

Efectos de sonido

En Vircon32 existen 16 canales para reproducir sonidos. Tenemos la opción de reproducir de manera automática (la función buscará un canal libre).

```
// reproducimos un sonido en cualquier canal libre
play_sound( SoundExplosion );
```

Otra opción es reservar manualmente algunos canales de sonido. Por ejemplo, si tenemos un sólo canal para que hable nuestro personaje, nos aseguramos de que nunca pueden estar sonando 2 frases al mismo tiempo (lo cual sonaría extraño).

```
// detenemos cualquier sonido anterior del personaje
if( get_channel_state( ChannelPlayer ) != channel_stopped )
    stop_channel( ChannelPlayer );

// y ahora ya podemos reproducir un nuevo sonido
play_sound_in_channel( SoundHello, ChannelPlayer );
```

Música de fondo

Normalmente nos interesará reservar un canal determinado para la música. Así podemos pausar o continuar la música cuando se necesite. También es habitual que la música suene continuamente y se repita en bucle. Aquí mostramos las dos cosas:

```
// configuramos la musica para reproducirse en bucle
select_sound( MusicLevel1 );
set_sound_loop( true );

// ahora la musica ya se reproducira en bucle
play_sound_in_channel( MusicLevel1, ChannelMusic );

// detenemos el canal de la musica
stop_channel( ChannelMusic );
```

Manejar objetos del juego

Cuando se tiene un juego más complejo, puede existir gran variedad de objetos que se tienen que actualizar y dibujar cada frame: jugador, disparos, varios tipos de enemigos, etc. En C++ se suele utilizar la herencia de clases para manejar estas entidades del juego de una forma más genérica. En C para esto podemos usar las uniones. En este pequeño ejemplo vemos cómo podemos definir objetos con datos variables:

```
// definimos estructuras con datos distintos
// dependiente de lo que necesite cada objeto
struct EnemyState{ ... };
struct BulletState{ ... };

// esta union puede guardar datos de cualquier objeto
union ObjectState
{
    EnemyState AsEnemy;
    BulletState AsBullet;
};

// a la union le añadimos informacion para saber de que se trata
struct GameObject
{
    bool Active;
    int ObjectType;
    ObjectData State;
}
```

Después debemos ver cómo manejar esos datos. Actualmente el compilador de Vircon32 es aún limitado y aún no hay soporte para memoria dinámica (con lo cual no podemos usar listas), ni punteros a funciones. Sin embargo sí que podemos manejar nuestros objetos a un nivel más genérico usando un array como mostramos aquí:

```
// usamos un array con el maximo numero de objetos que puedan existir
GameObject[ 50 ] AllObjects;

// asi procesariamos los objetos
for( int i = 0; i < 50; i++ )
{
    GameObject* ThisObject = GameObject[ i ];

    if( !GameObject[ i ]->Active )
        continue;

    // elegimos tipo de objeto
    if( ObjectType == TypeEnemy )
        GameObject[ i ]->Active = ProcessEnemy( GameObject[ i ]->AsEnemy );

    else if( ObjectType == TypeBullet )
        GameObject[ i ]->Active = ProcessBullet( GameObject[ i ]->AsBullet );
}
```

Dónde seguir aprendiendo

Con esto hemos visto lo básico que se necesita para crear un juego, pero sigue habiendo mucho por aprender. Para practicar lo que hemos visto aquí, se puede empezar por compilar los programas de los tutoriales e ir haciendo experimentos con ellos.

Más adelante, una buena forma de avanzar y tener nuevas ideas es ver cómo se han hecho otros juegos y programas para Vircon32. El código de los programas de test que hay disponible puede servir como ejemplos más avanzados, ya que también tienen bastantes comentarios y suelen seguir la estructura que mostramos aquí.

También es recomendable para más adelante conocer en detalle cómo funcionan los distintos sistemas de la consola. Esto lo explica la guía correspondiente de esta misma documentación.