Vircon32: Cómo funciona la consola

Documento con fecha 2023.01.22 Escrito por Carra

¿Qué es esto?

Este documento es una guía rápida para conocer el funcionamiento de la consola virtual Vircon32. No pretende ser una especificación ni cubrir todas las características de forma detallada: su objetivo es solamente servir de apoyo a quien quiera crear programas para la consola. Esta guía está orientada a programar Vircon32 en lenguaje C. Para crear programas en ensamblador hace falta conocer más detalles internos del funcionamiento que quedan fuera del alcance de este documento.

Vircon32

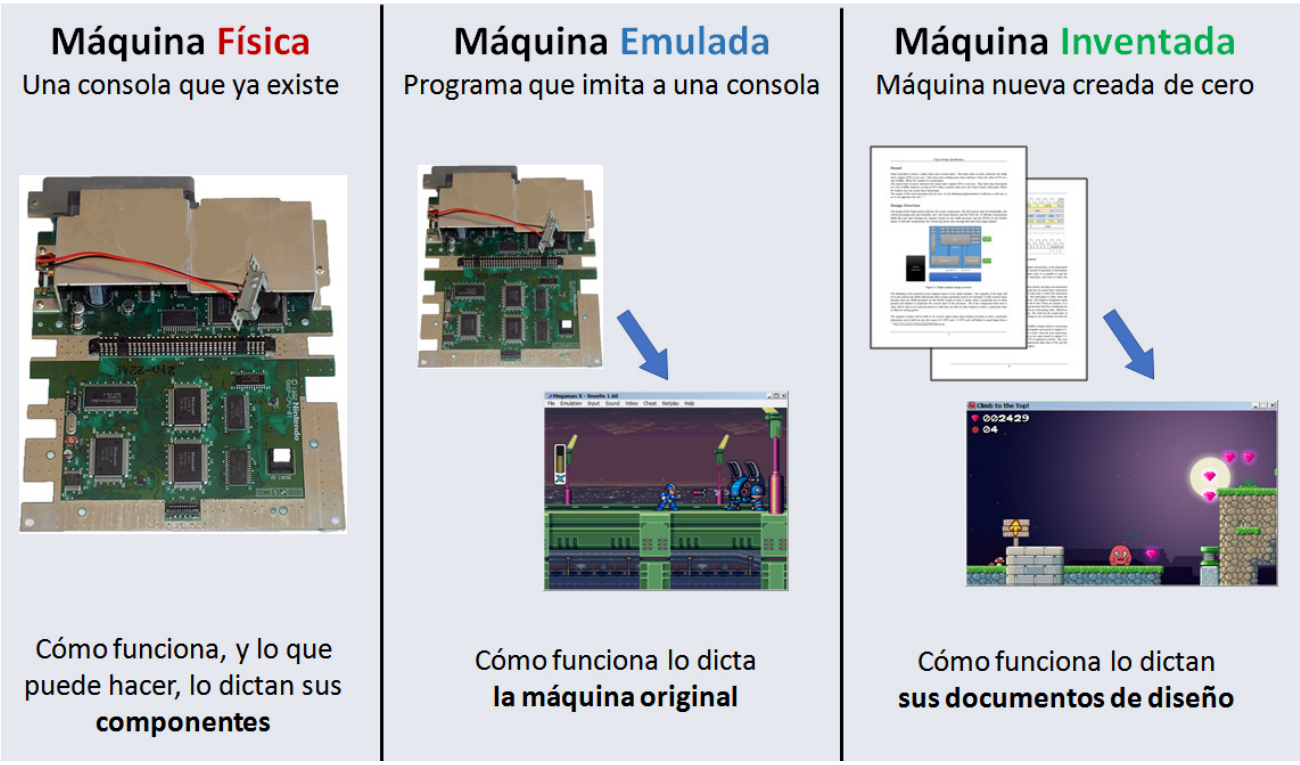
Índice

La primera parte de este documento explica las características generales de la consola y su arquitectura básica. En la parte segunda se cubre el funcionamiento de cada uno de los componentes de la consola.

PARTE 1: CONOCIENDO LA CONSOLA

Introducción

Vircon32 es una consola virtual de 32 bits. Esta máquina es inventada: no se basa en ninguna otra consola existente. Los documentos de diseño y las implementaciones que los cumplen SON la consola.



El proyecto completo cubre además otras áreas. Junto con el emulador y sus juegos van a existir además herramientas de desarrollo, guías como esta, programas de test, etc.

¿Cómo es Vircon32?

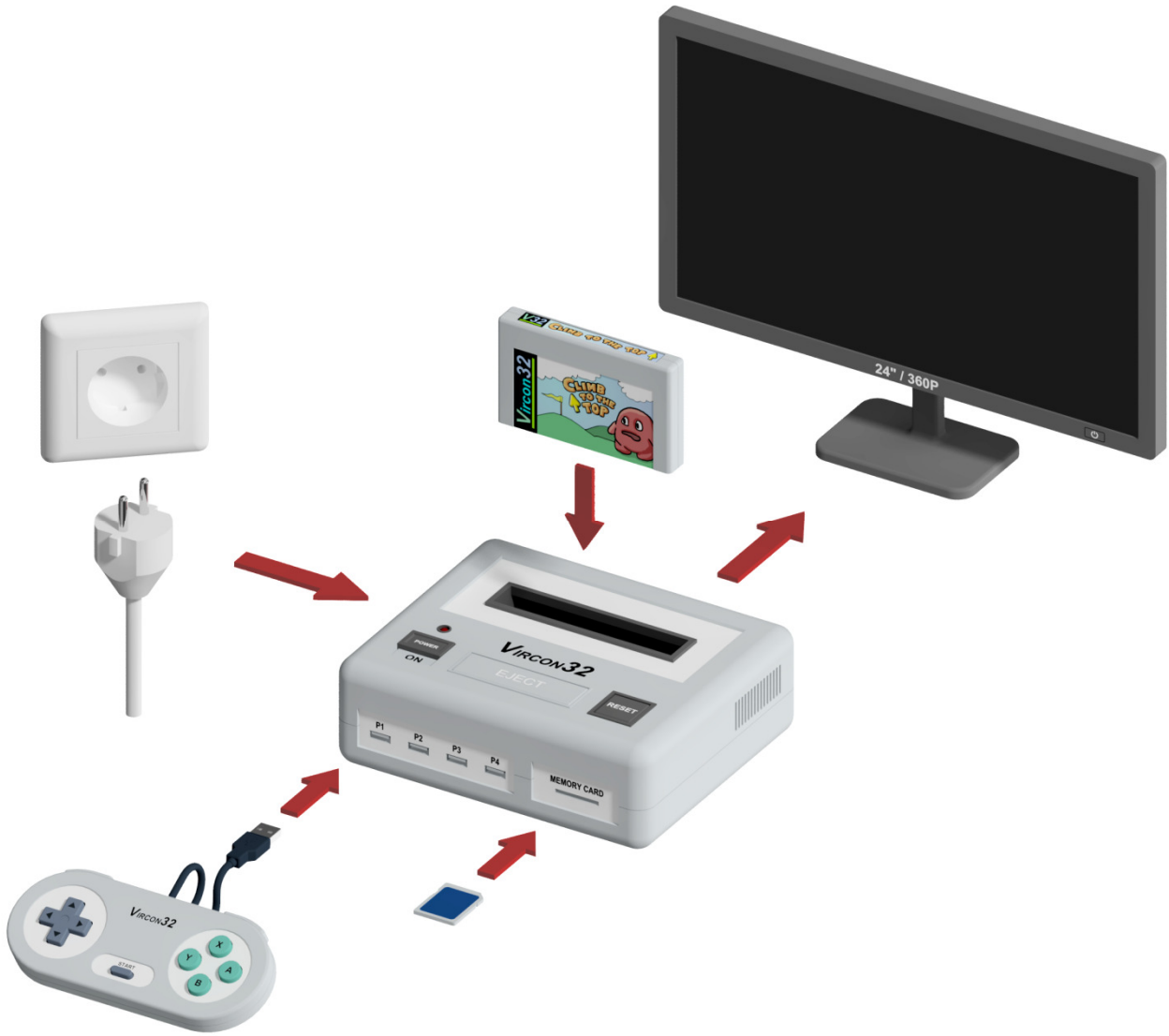
Vircon32 se ha diseñado con el objetivo de ser lo más sencilla posible. Está modelada como una máquina real (es decir: tiene su procesador, chip gráfico, controladores, buses…) pero sus componentes están simplificados para facilitar la programación de juegos, y también la creación de emuladores y otras herramientas.

El segundo objetivo a la hora de diseñar esta máquina ha sido permitir que los juegos que se hagan para ella puedan ser juegos elaborados, que vayan más allá de simples demos técnicas o minijuegos. Esta consola está basada en la generación de consolas de 32 bits, y

sus capacidades van acorde con aquella generación (aunque sin capacidades 3D). Así, permite ir más allá del típico “estilo 8 bits” que hoy en día es excesivamente limitado.

Partes de la consola

Vircon32 no sólo modela la consola, sino que se ocupa del sistema completo. Cosas como los mandos, la pantalla o el cartucho también tienen que estar incluídas en el diseño y las implementaciones deben tenerlos en cuenta.



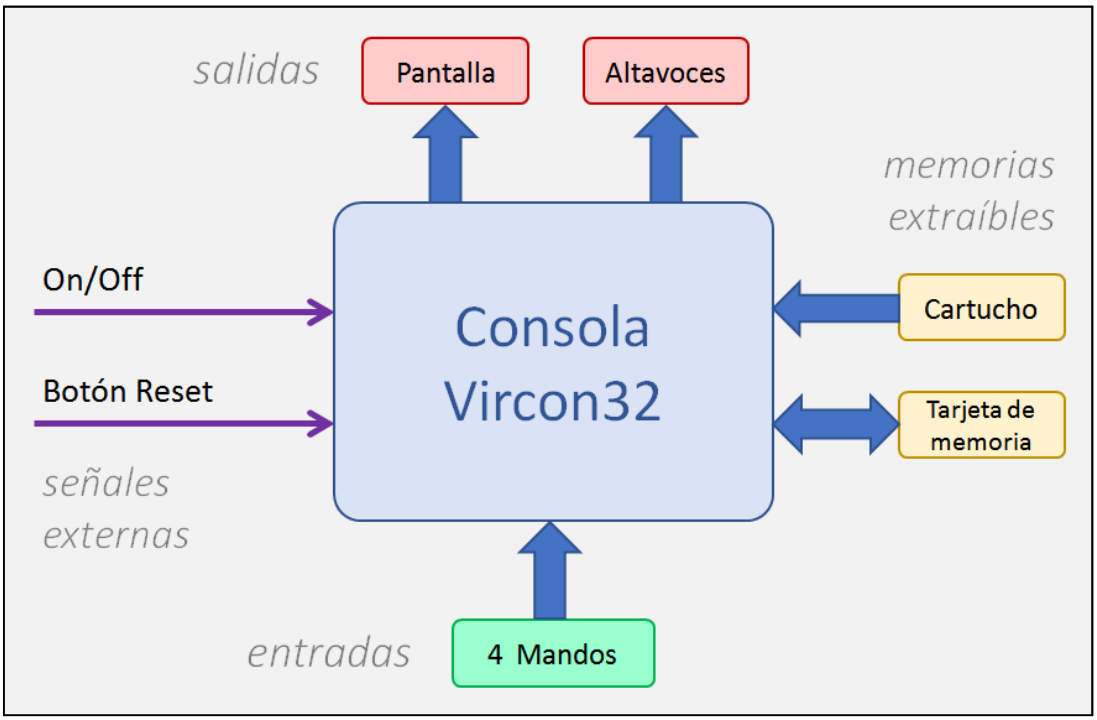
¿Qué significa esto? Tomemos por ejemplo el mando. En la realidad el jugador puede estar jugando con un tipo de mando distinto al de Vircon32, o incluso un teclado. Pero de cara a quien programa un juego, el mando con el que se está jugando siempre es el de Vircon32. Será tarea del emulador adaptar el control real, y al jugador le interesará intentar mapear sus botones de forma parecida a la disposición del mando de Vircon32 (ya que los juegos pueden dar por hecha esa colocación en sus controles).

En sí, el sistema Vircon32 no es muy distinto de las consolas clásicas de 32 bits: además de la consola en sí tenemos hasta 4 mandos, un cartucho, y opcionalmente una tarjeta de memoria. La salida de la consola es algún tipo de display de imagen y sonido.

La consola

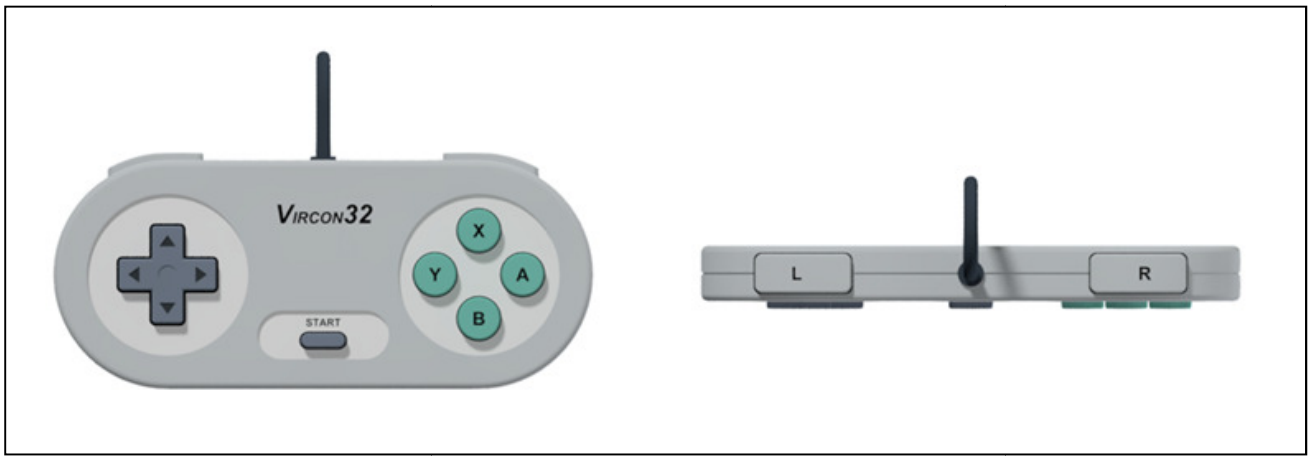
La consola es el componente central porque ejecuta los juegos. Pero su funcionamiento interno lo veremos en las secciones siguientes. A nivel de conexiones externas tenemos la entrada de cartucho, 4 puertos para mandos, ranura para tarjeta de memoria y su conexión a la pantalla.

Nosotros (los usuarios), además de conectar ó desconectar dispositivos a la consola, podemos controlar su funcionamiento con el interruptor de encendido y el botón de reset.



Los mandos

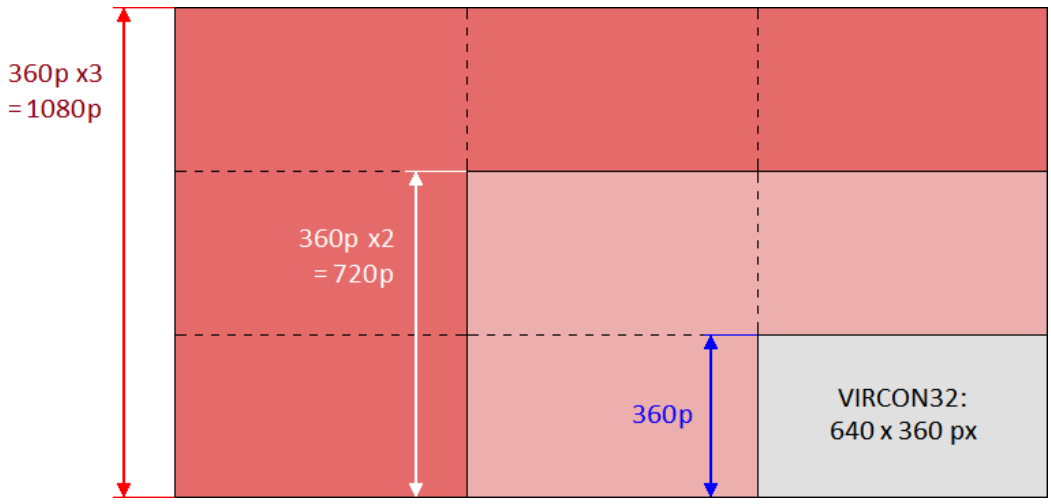
Los mandos de Vircon32 tienen una cruceta, 4 botones frontales, 2 botones superiores y un botón central de Start. La colocación de los elementos es la que se ve en esta imagen.



Todos los elementos del mando son digitales (sólo distinguen entre pulsados y no pulsados). La cruceta es de tipo basculante, con lo que en cada eje nunca pueden estar pulsadas a la vez las 2 direcciones opuestas.

La pantalla

La pantalla de Vircon32 tiene una resolución de 640x360 pixels a 16:9, y funciona a 60 fotogramas por segundo. Es decir, en términos actuales, es una resolución de 360p. Esta resolución nos permite adaptar la imagen de Vircon32 a las pantallas más usadas hoy en día (720p, 1080p, 1440p, 4K) mediante escalado exacto y así evitar deformaciones.



En cuanto a profundidad de color, la pantalla puede mostrar color verdadero (color RGB con 24 bits por pixel).

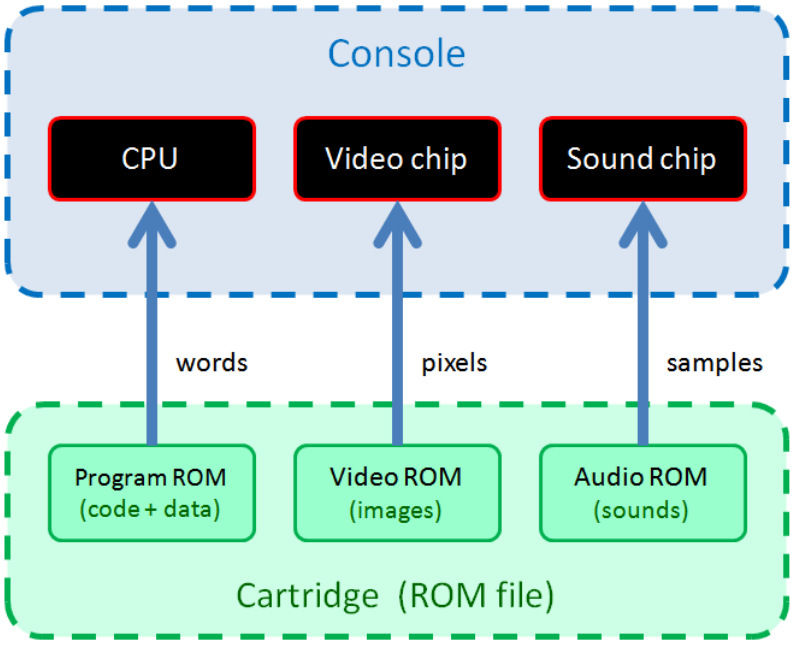
El cartucho

Los cartuchos de Vircon32 se componen de 3 memorias de sólo lectura (ROMs), que son independientes entre sí. Al insertar el cartucho, cada una de esas partes queda conectada a un componente de la consola:

- **La ROM de programa:** Contiene el programa a ejecutar y los datos que necesite ese programa. Se conecta al bus de memoria, desde donde la CPU puede acceder a ella.

- **La ROM de video:** Contiene una colección de todas las imágenes que puede mostrar el juego. Se conecta directamente al chip gráfico (la GPU).

- **La ROM de audio:** Contiene una colección de todos los sonidos que puede reproducir el juego. Se conecta directamente al chip de sonido (la SPU).

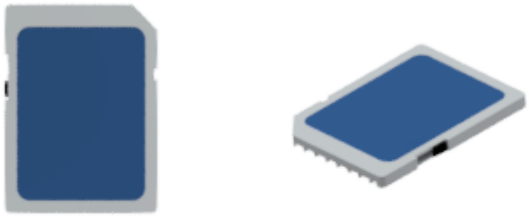


La consola puede leer esos 3 tipos de contenidos de los cartuchos de manera inmediata, con lo que no hay tiempos de carga ni es necesario copiar nada en memoria. Esto es posible porque, para mantener la simplicidad, las imágenes y sonidos se almacenan sin comprimir.

Sin embargo, al no haber ninguna compresión, los juegos necesitan poder ocupar más espacio. Para que esto no impida hacer juegos elaborados, los límites del cartucho se han definido de forma generosa. Las ROMs de audio y video pueden llegar a un máximo de 1GB cada una, mientras que la ROM de programa tiene un límite de 512 MB. Por tanto un cartucho de Vircon32 puede tener hasta 2.5GB de capacidad.

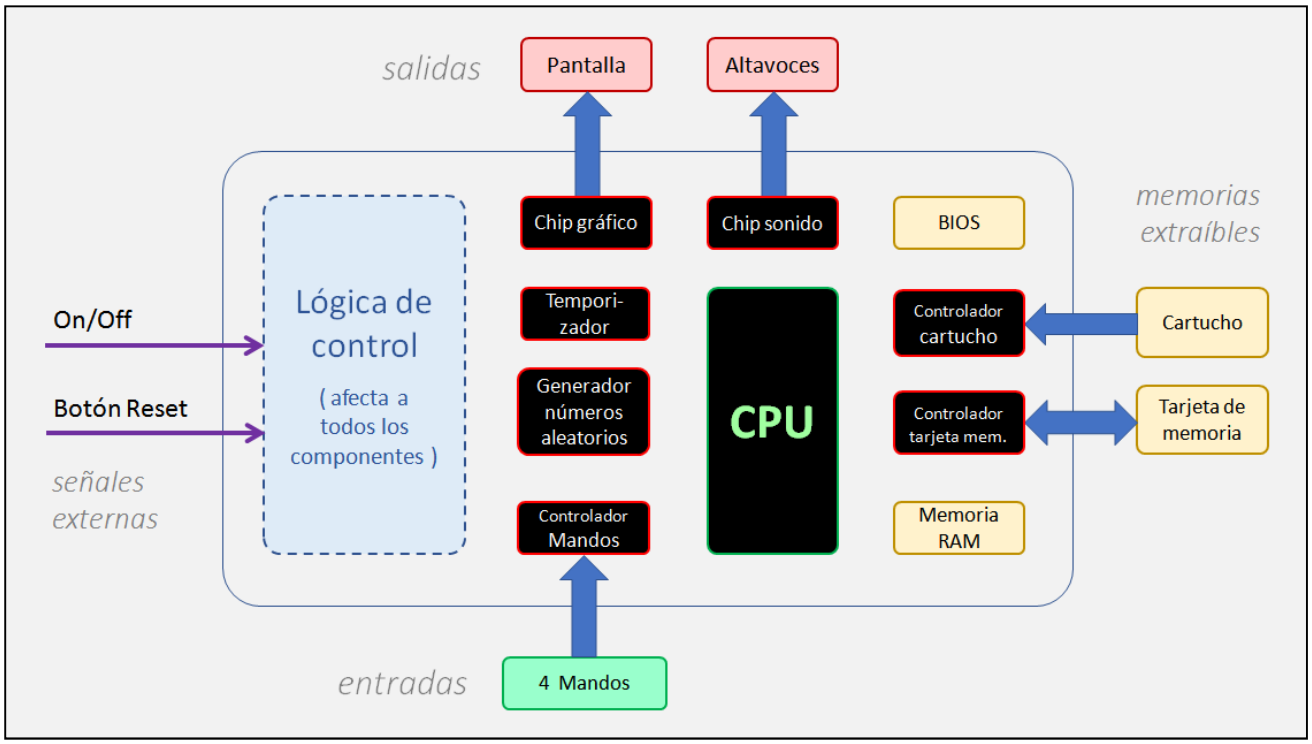
La tarjeta de memoria

Ya que los cartuchos son solamente ROMs (es decir, son de sólo lectura), la consola permite usar tarjetas de memoria como almacenamiento permanente. De esta forma permitimos que puedan existir juegos largos ó más complejos, en los que se pueden guardar partidas. La capacidad de estas tarjetas es de 1MB.



Arquitectura de la consola

Ya hemos explicado lo básico de los dispositivos que se puede conectar a la consola. Ahora vamos a ver con más detalle la propia consola. Podemos tomar el anterior esquema de conexiones del sistema completo, y ampliarlo para ver qué hay dentro de la consola. A un nivel básico, esto es lo que veríamos:



No hace falta conocer todo este detalle para programar juegos en C. Lo más importante es saber que la CPU es el chip principal desde el que ejecutamos el juego. La CPU tiene la capacidad de comunicarse con los demás chips para pedirles que realicen sus funciones

específicas. Por ejemplo, cuando el programa de nuestro juego llama a funciones que dibujan en pantalla, la CPU enviará al chip gráfico comandos que le indicarán qué debe dibujar y dónde.

Componentes que forman la consola

Para empezar a conocer los componentes de la consola vamos a dar una idea general sobre cada uno de ellos y sus funciones.

- **Procesador (CPU):** Es el chip principal, que ejecuta el programa del juego. El procesador ejecuta las instrucciones de nuestro programa una tras otra e interactúa con los demás chips cuando lo requiere.

- **Temporizador:** Controla cuándo debe suceder cada ciclo de ejecución de la CPU. También, 60 veces por segundo, inicia un nuevo frame. Esto activa el refresco de pantalla, pero también gobierna algunas acciones de otros chips.

- **Controlador de mandos:** Detecta qué mandos están conectados en cada momento y permite consultar el estado de sus crucetas y botones.

- **Chip gráfico (GPU):** Este chip accede a las imágenes presentes en la ROM de video del cartucho y las utiliza para dibujar en pantalla, pudiendo aplicarles varios efectos como rotaciones o zoom.

- **Chip de sonido (SPU):** La SPU tiene conexión directa con los sonidos en la ROM de audio del cartucho. Puede reproducir hasta 16 de ellos al mismo tiempo, y aplicar algunos efectos como cambios de velocidad y bucles.

- **Controlador de cartucho:** Detecta cuándo hay un cartucho conectado, y permite obtener información básica sobre su contenido. A través de él, la CPU puede acceder a la memoria de programa del cartucho.

- **Controlador de tarjeta de memoria:** Nos permite saber si hay una tarjeta de memoria conectada. Permite a la CPU acceder a la memoria de la tarjeta.

- **Generador de números aleatorios:** Usa un algoritmo para producir números pseudoaleatorios cada vez que se lo pedimos. Así nos permite simular el azar.

- **Memoria RAM:** Es la memoria de trabajo del programa que corre en la CPU.

- **La BIOS:** Es un software interno, similar al de un cartucho (programa, video y audio). Dirige el arranque de la consola y reacciona ante posibles errores de hardware. También incluye una fuente de texto para poder escribir en pantalla.

En la parte 2 de este documento veremos con detalle cada uno de los componentes de la consola, para conocer cómo funcionan y saber lo que pueden hacer nuestros programas.

Comunicación entre componentes

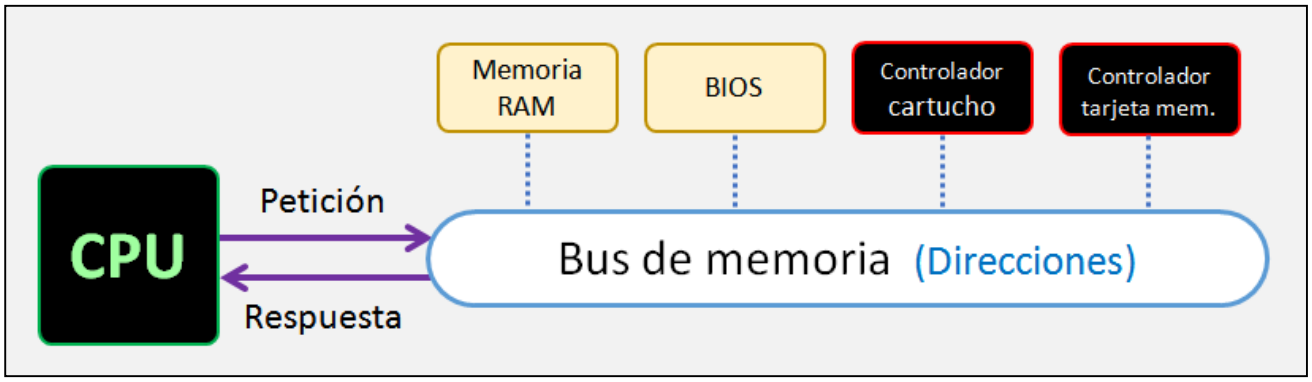
Los componentes de la consola que hemos descrito arriba no pueden funcionar de manera aislada. Deben ser capaces de comunicarse para poder interactuar y hacer que la consola funcione. Para esto existen 2 canales de comunicaciones o buses, que se usan con fines diferentes.

Los buses de Vircon32 usan un sistema maestro-esclavo, en el que hay un único dispositivo maestro que controla la comunicación y envía comandos. Los demás componentes conectados (los esclavos) actúan pasivamente y se limitan a contestar a las peticiones que reciben. En ambas buses la CPU es el dispositivo que controla la comunicación (el maestro), y la misión de los demás componentes conectados al bus es dar servicio al procesador.

Bus de memoria

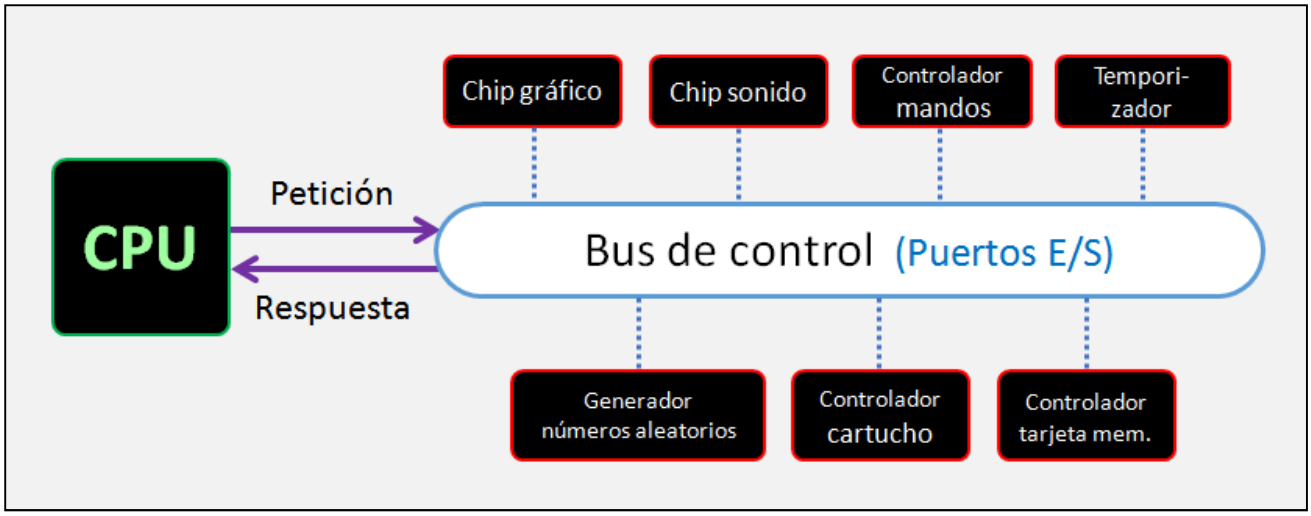
Este es el bus al que se conectan todos los dispositivos que tienen memoria (ya sea RAM o ROM). Al conectarse a este bus, su espacio de memoria recibe un rango de direcciones y su contenido se vuelve accesible por la CPU.

La CPU, como maestro, puede pedir al bus hacer una operación de lectura ó escritura en una determinada dirección de memoria. Si la operación se realiza, la CPU recibirá la respuesta con el resultado. Si no se puede realizar, se activará un error de hardware que detendrá nuestro programa. Esto puede ocurrir bien por no existir esa dirección, o porque en esa dirección la operación no es válida (por ejemplo: se ha intentado escribir en la memoria del cartucho, que es de sólo lectura).



Bus de control

Este bus sirve al procesador para controlar las funciones de otros chips y enviarles comandos para que actúen. El mecanismo de control es este: cada chip a controlar expone al exterior una serie de puertos de entrada/salida en los que se puede leer y/o escribir un valor. Al conectarse al bus de control, a cada chip se le asigna un rango de números de puerto y la CPU puede enviar y recibir datos de ellos.



En todos los puertos el valor que se puede enviar o recibir es de una sola palabra de 32 bits. Es decir, a nivel del bus manejar un puerto es muy similar a manejar una dirección de memoria. Y al igual que en el bus de memoria, si la CPU pide una operación que no se puede realizar, nuestro programa se detendrá por un error de hardware.

Sin embargo el comportamiento de cada uno de estos puertos puede ser muy distinto en cada caso. Existen puertos de lectura/escritura, de sólo lectura y de sólo escritura. Y mientras algunos puertos actúan como simples registros, en otros la operación de leer o escribir en el puerto desencadena otras acciones.

PARTE 2: COMPONENTES DE LA CONSOLA

En esta segunda parte veremos más a fondo cada componente de la consola: qué puede hacer y cómo podemos usarlo. Sin embargo antes vamos a dar un detalle importante que afecta a toda la consola en general: para simplificar la CPU y la memoria, en Vircon32 sólo se pueden manejar datos en unidades de 32 bits. Es decir, la unidad mínima en esta consola no es el byte, sino que es la palabra de 32 bits. A la hora de crear programas esto nos va a afectar de varias maneras:

- Los registros de la CPU siempre se usan al completo: no existen variantes de las instrucciones para manejar sólo 8 bits ó 16 bits como en otras CPUs.
- Las posiciones de memoria y sus offsets se dan siempre en palabras, no en bytes.
- Lo mismo ocurre con los tamaños de datos, pues en Vircon32 no es posible tener bytes individuales.

Sin embargo esto también nos da ciertas ventajas. Al programar otros sistemas debemos tener en cuenta la alineación de los bytes en memoria, y si se están agrupando los bytes en sistema big-endian ó little-endian. En Vircon32 esto no nos afecta.

Procesador (CPU)

La CPU es el componente central de la consola, ya que es el que ejecuta nuestro programa. Para crear juegos en Vircon32 no necesitamos saber mucho sobre la propia CPU, a no ser que vayamos a escribir código en ensamblador, así que vamos a dar solamente detalles generales.

Control de la CPU

Desde nuestros programas en C podemos usar algunas funciones para controlar la ejecución del programa. Podemos usar `end_frame()` para detener la CPU hasta que se inicie el siguiente frame. Esto nos sirve para controlar la velocidad del juego.

```
// bucle principal del juego
while( true )
{
    // procesamos este frame
    // (...)

    // esperamos al siguiente
    end_frame();
}
```

Por otra parte, si nuestro juego ha terminado y queremos detener el programa por completo tenemos 2 opciones. La primera de ellas es salir de la función main. La segunda es usar la función `exit()`, que se puede usar desde cualquier parte del código.

```
void ShowGameEnding()
{
    // mostrar secuencia final
    // (...)

    // el juego ha terminado y lo detenemos
    exit();
}
```

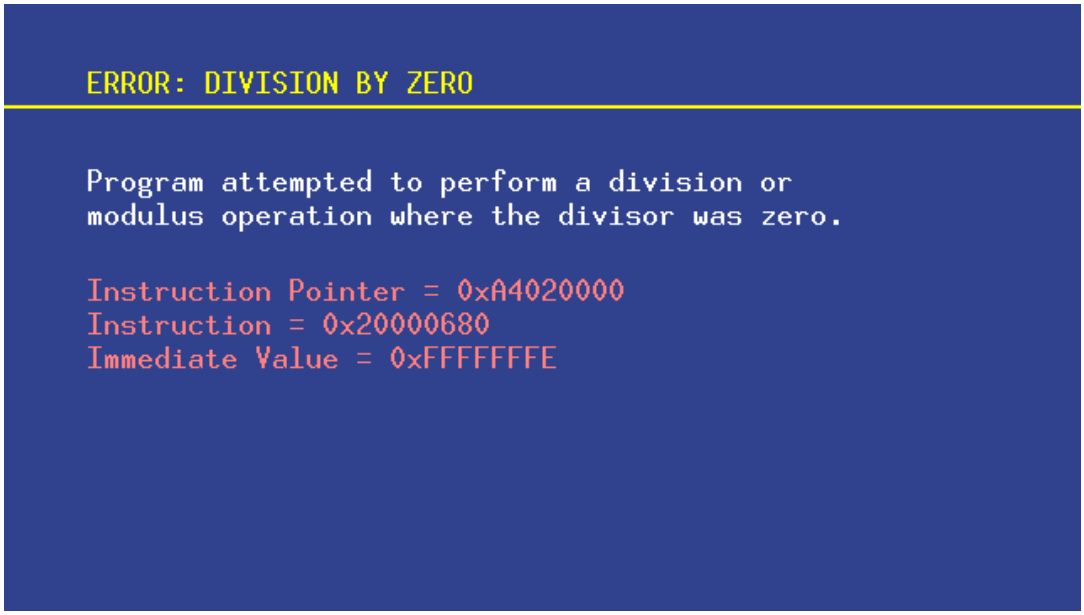
Rendimiento

La CPU funciona a 15 MHz. Por simplicidad esta CPU siempre ejecuta cualquier instrucción en 1 ciclo de reloj. En Vircon32 muchos aspectos de un juego se van a procesar para cada fotograma, y funcionando a 60fps eso nos da una capacidad de hasta 250.000 instrucciones por fotograma.

En teoría un juego también puede controlar la ejecución de tal forma que use más capacidad de procesado y no se actualice todos los frames sino por ejemplo cada 2 frames (a 30 fps). Sin embargo otras funciones como el refresco de la pantalla o la actualización de los mandos son independientes del programa y seguirán funcionando a 60 fps.

Errores hardware

Existen algunas situaciones al ejecutar un programa que pueden causar un error de hardware, tales como dividir por cero. Cualquiera de ellas hará que nuestro programa se detenga, y la consola mostrará un mensaje de error como el siguiente.



Temporizador

El temporizador lleva la cuenta del tiempo que pasa, y se ocupa de dar las señales adecuadas a otros componentes cuando deben realizar alguna acción. La medición del tiempo se hace a 3 niveles distintos:

Contador de ciclos

Este contador mide los ciclos de CPU que han pasado desde el inicio del frame. Es decir, cada frame cuenta desde 0 hasta 249.999. Lo podemos consultar desde C usando la función `get_cycle_counter()`. No es habitual que un juego lo use, pero puede tener algunos usos. Por ejemplo se puede leer el valor antes de terminar el frame para saber internamente la carga de CPU y mostrarla en pantalla en un modo debug.

```
// calculamos el % de CPU usado
int UsedCycles = get_cycle_counter();
float CPUPercentage = 100.0 * UsedCycles / 250000;

// esperamos al siguiente frame
end_frame();
```

Sin embargo debemos tener en cuenta que el contador de ciclos no se puede usar para temporizar eventos. Un emulador de Vircon32 está obligado a garantizar que el inicio de cada frame sucede cuando debe (a intervalos regulares, 60 veces por segundo), pero en cambio es libre de emular los ciclos de ese frame al ritmo que considere oportuno. Aunque leamos un contador de ciclos igual a 125.000, no podemos pensar que ha pasado exactamente 1/120 de segundo en tiempo real.

Contador de frames

El contador de frames se inicia a 0 cuando la consola se enciende, y mide el número de frames que han pasado desde ese momento. Este contador es el que más se usa en los programas, ya que es nuestra principal manera de medir tiempos. En este caso se usa la función `get_frame_counter()`. Por ejemplo, en el bucle principal podríamos hacer:

```
// guardamos el tiempo desde un evento inicial
if( Event.Happened )
  StartingFrames = get_frame_counter();

// (...)
// activamos una respuesta al evento 1 segundo despues
if( Event.Happened )
  if( get_frame_counter() - StartingFrames >= 60 )
    TriggerResponse( &Event );

// (...)
// esperamos al siguiente frame
end_frame();
```

Contamos además con la función `sleep()`, que se basa en el contador de frames para esperar un determinado tiempo. Es útil por ejemplo en transiciones de escenas en las que queremos una pequeña espera.

```
// fundimos la escena a negro
FadeToBlack( &Scene1 );

// mantenemos la pantalla negra 0.5 segundos
sleep( 30 );

// hacemos la transicion a la siguiente escena
FadeFromBlack( &Scene2 );
```

Fecha y hora

El temporizador no sólo controla el tiempo relativo a la propia consola, sino que también tiene un reloj interno con la fecha y hora. Éstas se pueden consultar con `get_date()` y `get_time()`. Estas funciones nos dan la información en el formato binario nativo de Vircon32 (un int para cada una), pero podemos convertirlas al formato humano normal de la siguiente forma:

```
// leer fecha y hora actual, en el formato nativo de Vircon
int VirconTime = get_time();
int VirconDate = get_date();

// traducir fecha a formato humano
// (estructura con año, mes y dia)
date_info HumanDate;
translate_date( VirconDate, &HumanDate );

// traducir hora a formato humano
// (estructura con horas, minutos y segundos)
time_info HumanTime;
translate_time( VirconTime, &HumanTime );
```

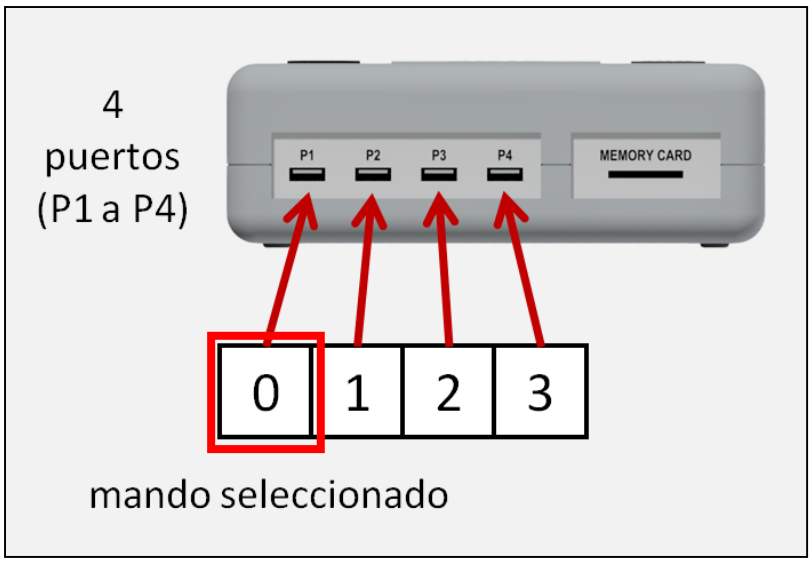
Un juego podría usar estos valores para almacenar la fecha y hora en la que se guardó una partida en la tarjeta de memoria, y mostrarla al jugador cuando la partida se cargue.

Controlador de mandos

Este controlador monitoriza constantemente el estado de los mandos conectados a cada uno de los 4 puertos de la consola. Además de darnos información sobre la cruceta y los botones, también nos indica qué mandos hay conectados (pues se puede conectar y desconectar mandos en cualquier momento).

Un concepto importante es que, aunque existen 4 mandos (conectados o no), en el controlador siempre está seleccionado solamente uno de ellos. Los 4 mandos siempre se

mantienen actualizados, pero nuestras consultas siempre se hacen sobre el mando que esté actualmente seleccionado. El mando seleccionado por defecto es el primero.



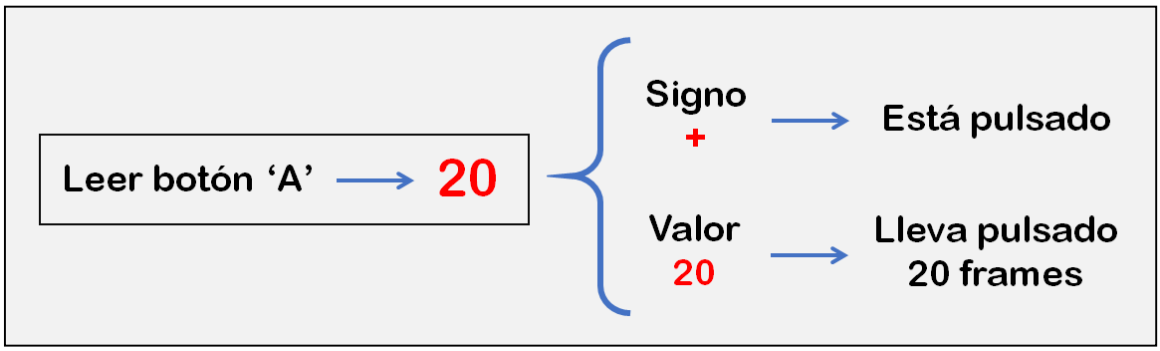
Desde nuestro programa podemos usar `select_gamepad()` para cambiar el mando seleccionado. Una vez que elijamos un mando podemos comprobar qué se está pulsando en él con las funciones gamepad_xxx, según la dirección o botón que queramos leer:

```
// leeremos estados del mando del jugador 2
select_gamepad( 1 );

// al pulsar el boton A, dispara
if( gamepad_button_a() == 1 )
  Shoot( &Player2 );

// si esta pulsando izquierda, retrocede
if( gamepad_left() > 0 )
  Player2.x -= 3;
```

En el mando de Vircon32 en sí, los estados de botones y direcciones son un simple sí o no (pulsados o soltados). Pero como se aprecia en el código, las funciones de consulta no nos devuelven un booleano, sino un entero. Esto es así porque el controlador de mandos nos da más información: podemos usar sólo el signo para saber si está pulsado, pero además el valor en sí nos dice cuánto tiempo lleva en el estado actual.



Es por eso que en el ejemplo, disparamos solamente cuando el valor es 1: el botón se acaba de pulsar este mismo frame. No seguimos disparando por cada frame pulsado. En cambio al mirar la dirección no estamos interesados en el tiempo y sólo comprobamos su signo, puesto que sí desplazamos al jugador todos los frames.

El estado que recibimos de un botón o dirección siempre tiene un signo: el controlador de mandos garantizará que nunca puede ser cero. Esto nos facilita las cosas al programar.

Refresco del estado de los mandos

El controlador no funciona con eventos sino que, al principio de cada frame, comprueba automáticamente las pulsaciones de todos los mandos y actualiza su estado. El programa no necesita hacer nada para ello: sólo consultar cuando lo requiera.

Chip gráfico (GPU)

La GPU es, salvando el procesador, el componente más complejo de Vircon32. Sin embargo es un chip gráfico muy simplificado. De hecho, solamente hay 2 tipos de comandos que la GPU puede ejecutar si el procesador se lo pide:

- Puede borrar la pantalla, con un color constante.
- Puede dibujar en pantalla una región de alguna de las texturas del cartucho.

Comandos sin parámetros

Los comandos de dibujo que puede hacer la GPU necesitan de algún parámetro. Por ejemplo, si vamos a dibujar parte de una textura en la pantalla necesitaremos saber de qué región hablamos y en qué punto de la pantalla la vamos a colocar. Si estos comandos se usaran como una función, el procesador invocaría un comando pasándole sus parámetros de una manera parecida a esta:

```
DibujarRegion( Region, X, Y )
```

Sin embargo la GPU recibe estos comandos como simples valores en uno de sus puertos, con lo cual los comandos no pueden recibir parámetros. Lo que ocurre es que en la GPU existen internamente una serie de parámetros configurables (que iremos viendo) y que los comandos utilizan a la hora de ejecutarse. Los programas deberán ajustar estos valores antes de enviar un comando. Así, la función de arriba en realidad tendría estas fases:

```
RegionSeleccionada = Region
X_PuntoDeDibujo = X
Y_PuntoDeDibujo = Y
DibujarRegion( )
```

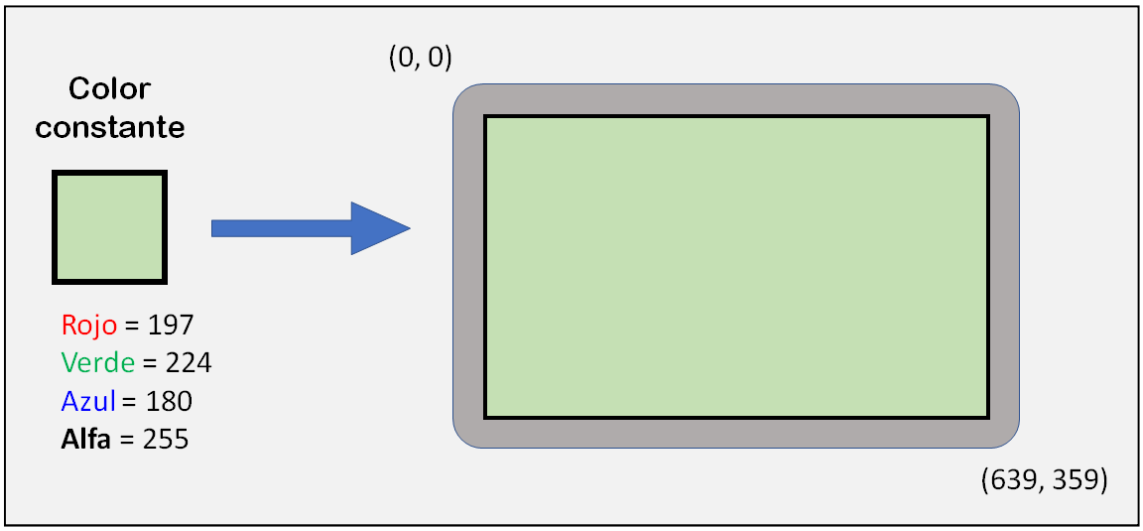
En algunos casos las librerías estándar de C tratan de facilitarnos el trabajo y nos permitirán usar comandos de GPU (u otros chips) como si fueran funciones. Por ejemplo, para dibujar una región en el centro de la pantalla las librerías nos permiten estas 2 alternativas:

```
// opcion 1: con algunos parametros ya integrados en el comando
select_region( RegionCharacter );
draw_region_at( 320, 240 );

// opcion 2: comando con todos sus parametros por separado
select_region( RegionCharacter );
set_drawing_point( 320, 240 );
draw_region();
```

Borrar la pantalla

Este comando es muy sencillo: sólo depende de un parámetro interno, que es el color de borrado. El comando aplicará este color de forma uniforme a toda la pantalla.



El color es un valor de 32 bits en formato RGBA. En las librerías de C ya existen algunos colores predefinidos. Por ejemplo, para borrar la pantalla en color azul haríamos:

```
clear_screen( color_blue );
```

Además de usar colores predefinidos también podemos escribir su valor numérico. Pero debemos tener en cuenta que en ese caso su representación escrita se invierte (ABGR) dado que Vircon32 almacena los bytes de cada palabra en sistema little-endian. Por ejemplo en un programa el color azul se escribiría como FFFF0000 (y no 0000FFFF). Una opción más fácil es usar funciones de la librería de C para crear colores:

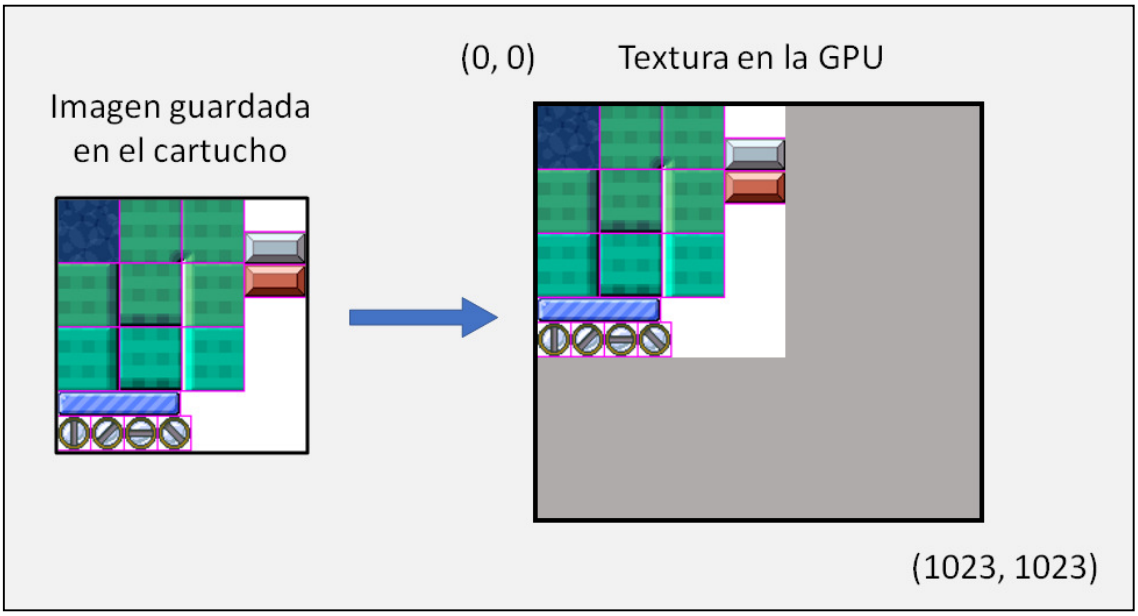
```
// tambien podemos usar make_color_rgba si queremos usar alfa
clear_screen( make_color_rgb( 0, 0, 255 ) );
```

En el borrado de pantalla puede utilizarse el canal alfa (opacidad) para hacer un borrado parcial. Si hacemos un borrado en negro pero con alfa = 128, oscureceremos la imagen en pantalla pero sin hacerla totalmente negra.

Texturas de la GPU

Para describir el segundo tipo de comandos de la GPU necesitamos conocer el concepto de textura. Como sabemos, la GPU trabaja con una serie de imágenes guardadas en el cartucho. La GPU solamente puede trabajar con imágenes de un tamaño estándar de 1024x1024 pixels, mientras que cada imagen del cartucho puede tener un tamaño diferente (hasta ese límite).

Para compensar esto, cuando el cartucho se conecta a la GPU, las imágenes del cartucho quedan almacenadas en la memoria de video colocándose como se ve en esta imagen. Al coincidir ambas en el primer pixel (0,0), las coordenadas de todos los pixels se conservan. A cada una de estas imágenes, ya almacenadas y adaptadas a la GPU, la llamamos una textura.

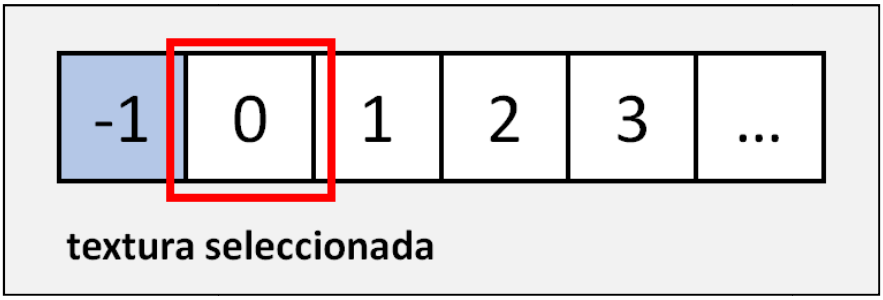


La lista de texturas

La GPU dispone de espacio para hasta 256 texturas en el cartucho, más 1 textura adicional para la BIOS. Todas estas texturas se manejan como una lista numerada: la textura de la BIOS (que siempre existe), recibe el ID especial -1. Las texturas que vienen del cartucho se numeran con IDs de 0 a 255.

La numeración sigue el mismo orden que las imágenes del cartucho: la primera que exista será la ID 0, la segunda la ID 1, etc. Los IDs que no se utilicen por el cartucho, hasta el 255, quedarán sin textura asociada y no se podrán usar en el programa.

Entre todas las texturas que existan, los comandos siempre se aplican sobre la textura que actualmente esté seleccionada en la GPU. Por defecto la textura seleccionada es la de la BIOS, ya que es la única que se garantiza que siempre exista (un cartucho puede no contener ninguna textura).



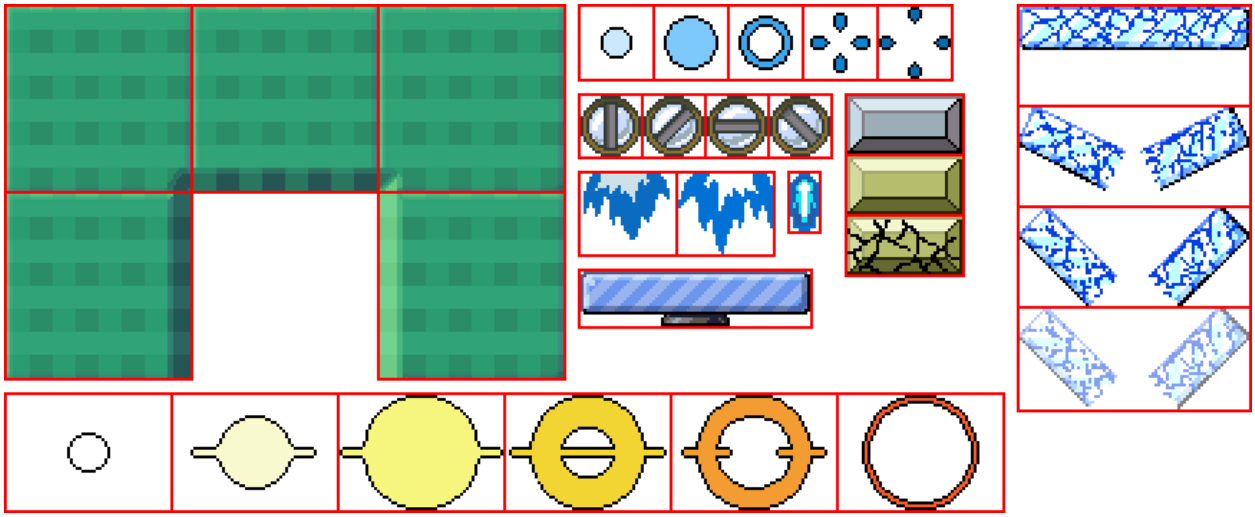
En nuestros programas podemos usar `select_texture()` para cambiarla. Si nuestro programa tiene varias texturas suele ser recomendable que les pongamos nombre.

```
#define TextureLevels  0
#define TextureMenus   1

// (...)
select_texture( TextureLevels );
```

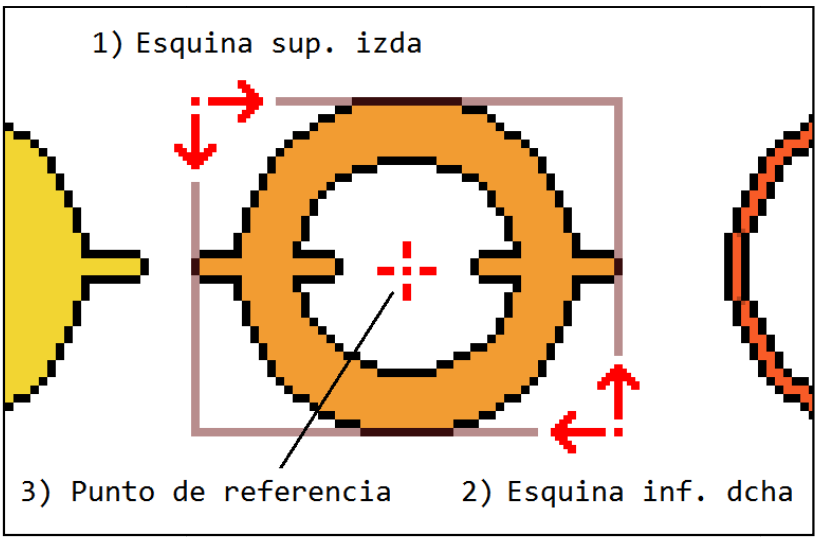
Regiones de cada textura

Las texturas de la GPU son más grandes que la propia pantalla. Por ello la GPU no dibuja directamente las texturas sino solamente regiones de la misma. Normalmente en una misma textura se almacenan juntas varias imágenes del juego, como en este ejemplo:

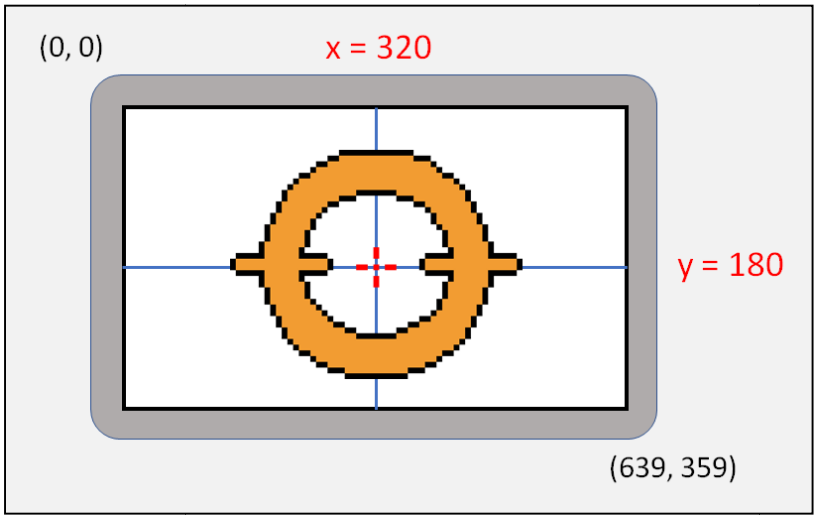


Los bordes separando las regiones no son realmente necesarios para la consola. Pero facilitan las cosas al programador a la hora de crear y usar las texturas.

En cada textura de la GPU existen 4096 regiones, con IDs de 0 a 4095. La manera de definir una región es usando 3 puntos de la textura, como se ve en la imagen siguiente:



Con los 2 primeros puntos delimitamos un rectángulo con la zona a dibujar. El tercer punto es el que se va a tomar como referencia para posicionar la imagen en pantalla. Por ejemplo, si estamos dibujando en el centro de la pantalla (punto 320,180), el punto de referencia de nuestra región se hará coincidir con esas coordenadas.



Al igual que existe siempre una ID de textura seleccionada, la GPU también guarda un ID de región seleccionada, que será la que utilicen los comandos. Podemos cambiar la región seleccionada usando `select_region()`. Sin embargo, debemos tener en cuenta que la ID de región seleccionada es un parámetro global de la GPU. No se guarda un ID de región seleccionada distinto para cada textura.

Nuestras funciones en C nos permiten definir regiones de varias maneras, pero la más elemental es `define_region()`, que aplica los 3 puntos que definen la región.

```
#define TextureLevels    0
#define RegionPlatform  50

// definimos la plataforma
select_texture( TextureLevels );
select_region( RegionPlatform );
define_region( 0,0,  324,49,  112,49 );  // referencia en punto central inferior
```

Como vemos, a diferencia de los IDs de texturas, los IDs de regiones los podemos elegir a voluntad. En cada textura siempre existen sus 4096 regiones, y nosotros simplemente las configuramos. No tenemos por qué empezar en el ID = 0, ni los IDs necesitan ser consecutivos.

Comandos para dibujar regiones

Como ya hemos visto, cuando la GPU dibuja en la pantalla alguna región siempre lo hace según le marquen sus parámetros internos. Es decir: usará la textura y región que estén actualmente seleccionadas, y además colocará la imagen usando otros 2 parámetros (coordenadas x, y) que son el punto actual de dibujado. Ya hemos visto en ejemplos anteriores que podemos cambiarlo con `set_drawing_point()`, o bien usar funciones que ya lo integran.

Sin embargo, no sólo se puede dibujar las regiones en pantalla tal y como son en sus texturas. La GPU también puede aplicarles transformaciones, en este caso rotación y/o escalado. Al dibujar una región podemos elegir aplicar o no cada una de estas 2 transformaciones, y para ello tenemos un total de 4 comandos. En esta tabla podemos ver cuáles son las funciones de C que se usan en cada caso:

Función	Escalado	Rotación
`draw_region()`	no	no
`draw_region_zoomed()`	sí	no
`draw_region_rotated()`	no	sí
`draw_region_rotozoomed()`	sí	sí

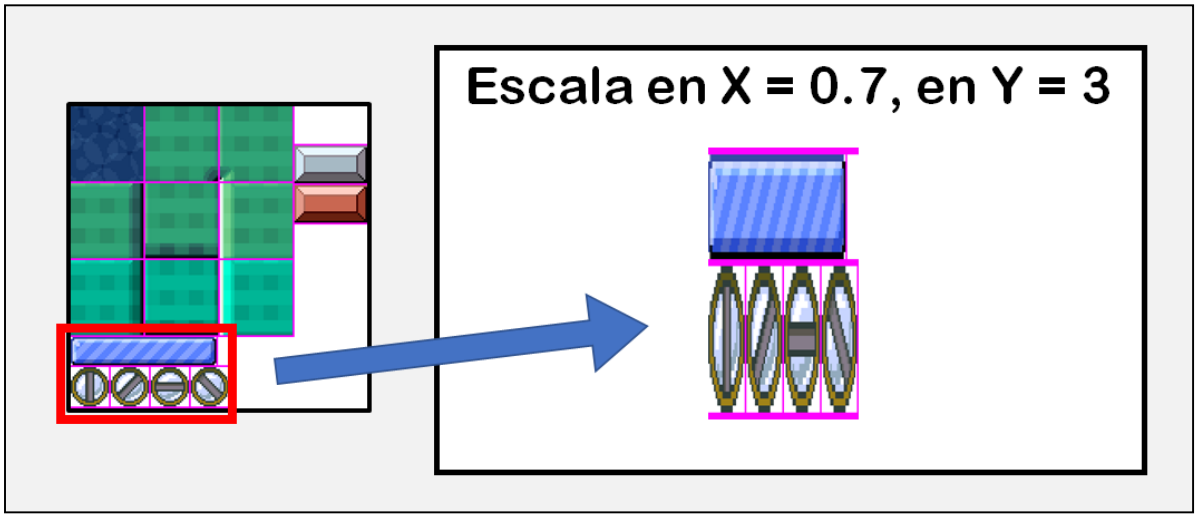
Cuando usamos un comando que no aplica alguna transformación, la GPU simplemente ignorará los parámetros asociados.

Dibujar con escalado

Cuando dibujamos una región aplicando el escalado, el comando usa dos parámetros adicionales que son los factores de escalado en X e Y. Estos factores, además de poder ser distintos entre sí, también pueden ser negativos. El efecto de un escalado negativo es que la imagen se invierte a lo largo de este eje. Por ejemplo, podríamos escribir esto:

```
// escalado distinto en X y en Y
select_region( RegionPlatform );
set_drawing_scale( 0.7, 3 );
draw_region_zoomed_at( 320, 180 );
```

Y el resultado que mostraría la GPU sería como este:

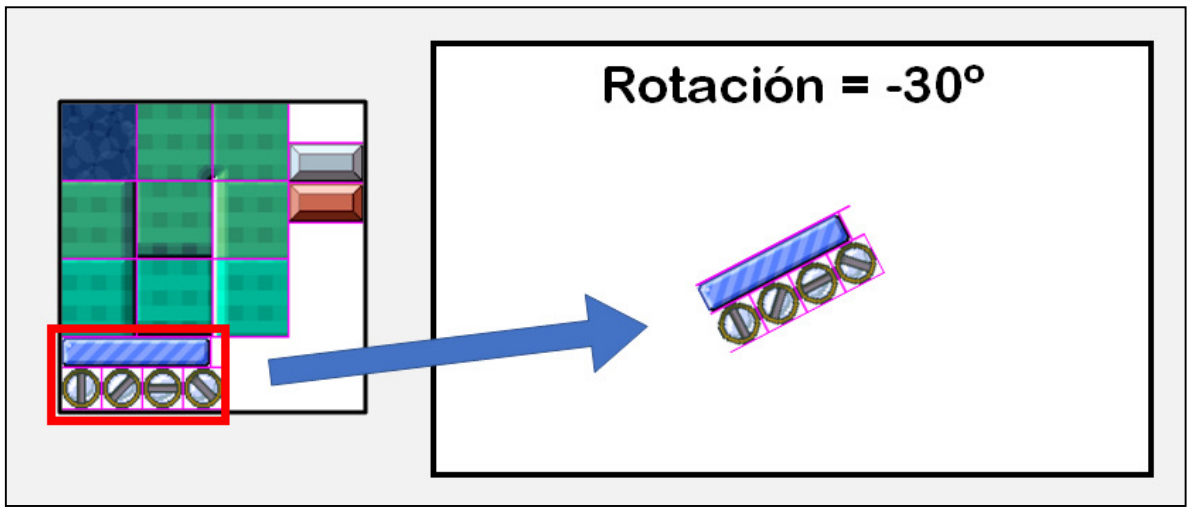


Dibujar con rotación

Si dibujamos una región usando rotación, el comando usará otro parámetro que es el ángulo de dibujado. Los ángulos en Vircon32 se miden siempre en radianes (180 grados son pi radianes). Además, como las coordenadas de pantalla comienzan arriba, el eje Y está invertido. Esto hace que los ángulos crezcan en sentido horario y no antihorario. Un ejemplo sería el siguiente:

```
// para rotar en sentido antihorario el angulo es negativo
select_region( RegionPlatform );
set_drawing_angle( -pi / 6 );   // pi = 180º, luego -pi/6 = -30º
draw_region_rotated_at( 320, 180 );
```

Y en ese caso en pantalla se nos mostraría lo siguiente. El centro de rotación siempre es el punto de referencia de la región. Es decir: aún con transformaciones, el punto de referencia siempre se dibuja coincidiendo con el punto de dibujado de la GPU.



Modificar los colores

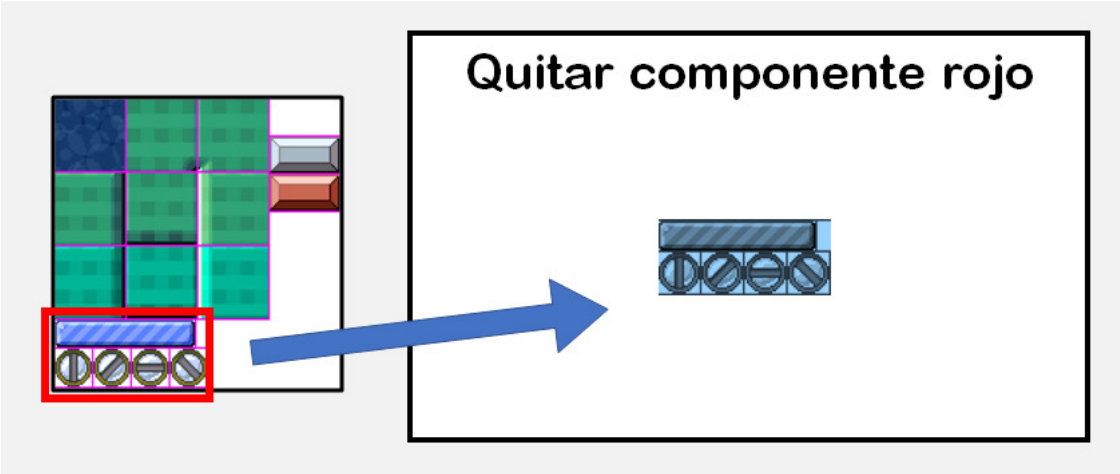
Cuando dibujamos regiones también podemos hacerlo aplicando una transformación a los colores. Existe un parámetro en la GPU llamado el "color de multiplicado", y que modifica los colores que se dibujan con un efecto de multiplicado de las componentes RGBA.

Para quien conozca OpenGL, el efecto es el mismo que el de glColor. Es decir, el color neutral es el blanco (el color de multiplicado por defecto) y según se oscurezcan las componentes RGB del color de multiplicado se oscurecerán las componentes de la región que se dibuja, actuando como un filtrado. También se multiplica la componente Alfa, lo que nos permite dibujar nuestra región con distintos niveles de opacidad.

Por ejemplo, podríamos escribir lo siguiente para eliminar el componente rojo de los colores de nuestra región.

```
// color de multiplicado con azul y verde pero sin rojo (en este caso, color cian)
select_region( RegionPlatform );
set_multiply_color( make_color_rgb( 0, 255, 255 ) );
draw_region_at( 320, 180 );
```

Y el efecto visual que veríamos en pantalla con este código es este:



El color de multiplicado se aplica en todos los comandos para dibujar regiones (las 4 variantes), pero no afecta al comando de borrar la pantalla.

Modos de mezclar colores

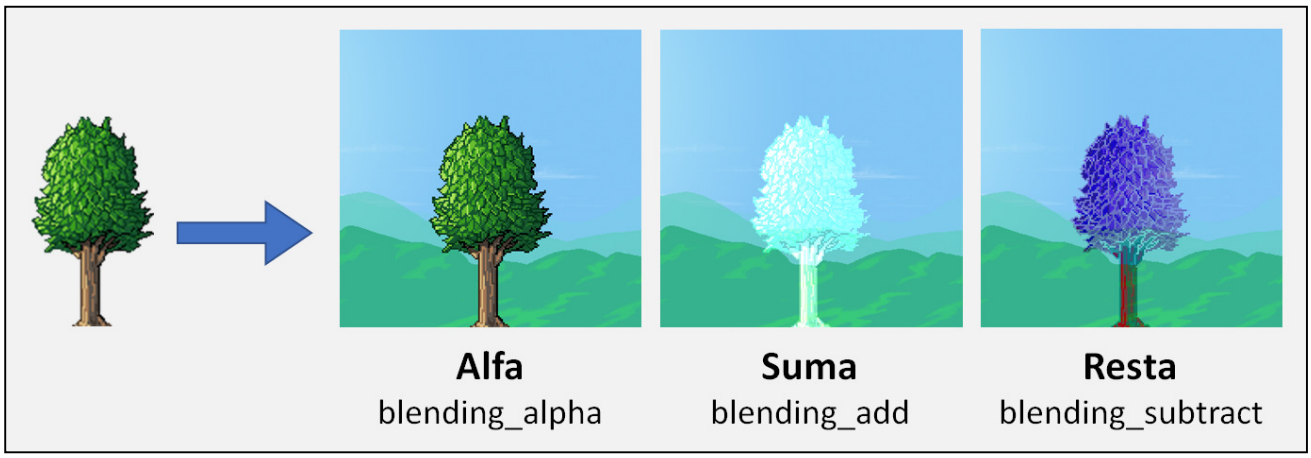
El mezclado de color es la manera en que se dibuja una imagen sobre un fondo. Para entender los efectos que se consiguen variando el mezclado, lo más sencillo es observar diferentes los diferentes tipos de capas en un programa de dibujo tipo Photoshop.

La GPU de Vircon32 dispone de 3 modos distintos de mezclado de color.

- **Modo Alfa:** Es el mezclado por defecto, y dibuja la región de la forma intuitiva: los colores no se modifican, y usa la componente Alfa de la región como grado de opacidad. El efecto es que podemos tener imágenes semitransparentes que se mezclan con el fondo sobre el que dibujamos.

- **Modo suma:** También conocido como "Linear Dodge" en programas de dibujo. Los colores de la región se suman a los del fondo, con lo que hace un efecto de iluminación. Al ser así, el negro es un color neutro.

- **Modo resta:** También se conoce como "Difference" en programas de dibujo. A los colores del fondo se les resta los de la región dibujada, pudiendo hacer efectos tipo sombreado. Esto hace que el negro también sea el color neutro en este caso.



Podemos cambiar el modo de mezclado usando `set_blending_mode()`, usando los nombres de los modos que figuran en la imagen anterior.

Si combinamos los diferentes modos podemos conseguir efectos ambientales como estos:

Los diferentes modos de mezclado, a diferencia de las modificaciones de color, no sólo se aplican a la hora de dibujar regiones sino que también afectan al borrado de pantalla.

Rendimiento de la GPU

El chip gráfico no estaría bien definido si no tuviera un límite de rendimiento determinado. Para esta GPU el rendimiento está basado en el número de pixels que se dibujan por cada frame.

Cada frame la GPU comienza con un contador de pixels restantes igual a 9 veces la pantalla completa (lo cual equivale a 1 pantalla completa de resolución 1080p). Cada vez que se realiza cualquier comando, se calcula una aproximación muy básica de cuántos pixels va a consumir. Si en algún momento la GPU no tiene capacidad suficiente en el presente frame, ignorará cualquier petición de realizar comandos hasta el frame siguiente.

El cálculo de los pixels consumidos por cada comando está pensado para ser rápido y simple, aún si no es muy preciso. No es necesario saber este proceso para crear juegos, pero puede hacer falta a quien pueda necesitar exprimir la GPU. El proceso es este:

- Se calcula el "ancho efectivo": la anchura en pixels de la pantalla de lo que se va a dibujar, aplicando el escalado en X si procede (sin signo). Si es mayor que el ancho de la pantalla, se recorta a 640 pixels.
- Se calcula el "alto efectivo": análogo al anterior. Si es mayor que el alto de la pantalla, se recorta a 360 pixels.
- El número de pixels consumidos se calcula como: "ancho efectivo" x "alto efectivo".
- No todas las operaciones son igual de costosas, con lo cual dependiendo de la operación realizada, se modifican los pixels consumidos con estos factores:
    - En un borrado de pantalla: -50%
    - En dibujo de regiones con escalado activado: +15%
    - En dibujo de regiones con rotación activada: +25%
      (si se aplican ambos efectos el factor es +40%)

Como se puede ver este cálculo ignora la posición de dibujado. Es decir, las partes de la región que puedan quedar fuera de la pantalla también cuentan como pixels usados. Tampoco influye qué mezclado de color se está usando. Se consideran igual de costosos.

Chip de sonido (SPU)

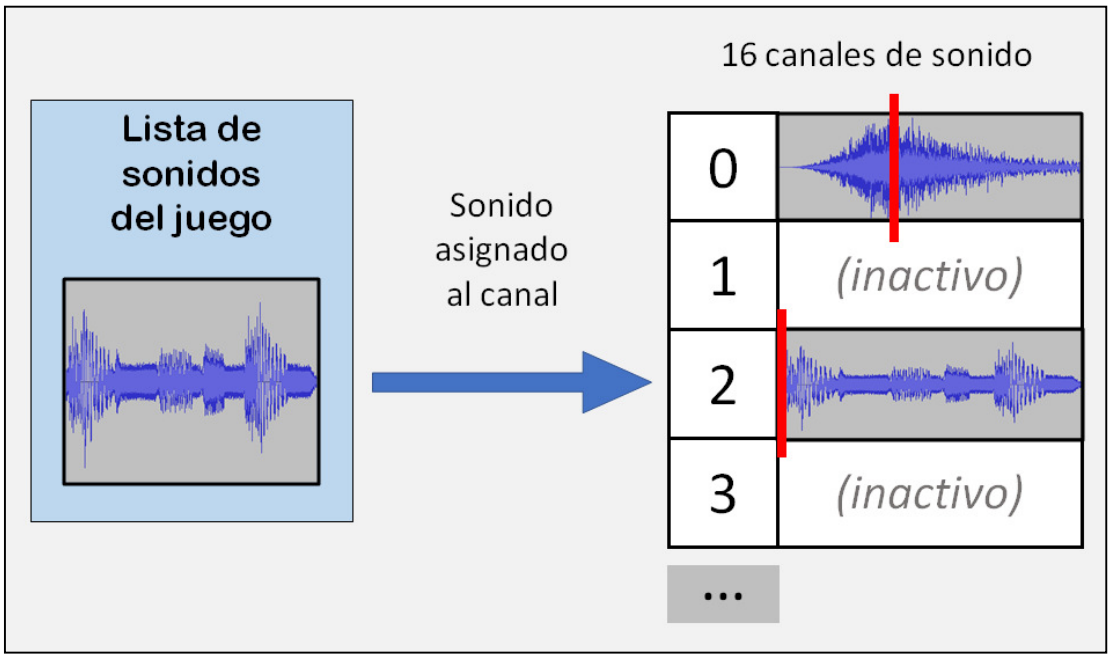
El chip de sonido tiene características comunes con la GPU, aunque es más sencillo. Al igual que la GPU, tiene una lista numerada de hasta 1024 sonidos para el cartucho (con sus IDs en el mismo orden, de 0 a 1023), más la ID = -1 reservada para el sonido de la BIOS.

En esa lista de sonidos siempre existe un sonido seleccionado que podemos cambiar usando `select_sound()` con la ID correspondiente. El sonido seleccionado por defecto es el de la BIOS, pues es el único que siempre existe (un cartucho puede no tener sonidos).

La diferencia con el chip gráfico es que los sonidos se utilizan tal cual. No necesitan tener un tamaño estándar, ni los reproducimos usando partes de ellos como pasaba con las regiones. En muchos casos no necesitaremos configurar nada en los sonidos.

Canales de reproducción

El chip de sonido dispone de 16 canales en los que podemos reproducir sonidos. Están numerados con las IDs de 0 a 15. Siempre existe un canal seleccionado en la SPU (por defecto el 0), que podemos cambiar con `select_channel()`. Cuando un canal está seleccionado le podemos asignar un sonido de la lista (con su ID) y empezar a reproducir.



Cada canal de sonido es independiente de los demás, y todos ellos pueden estar reproduciendo a la vez cualquiera de los sonidos. Pero debemos tener en cuenta que si muchos sonidos suenan a la vez, saturarán el altavoz a no ser que reduzcamos sus volúmenes.

Un canal puede estar en 3 estados: reproduciendo, pausado y parado. Cuando un canal no tiene sonido asignado o ya terminó de reproducir, su estado siempre es parado. El estado de un canal lo podemos consultar con `get_channel_state()`:

```
// esperamos a que un sonido termine
while( get_channel_state( ChannelSoundEffects ) != channel_stopped )
  end_frame();
```

Comandos de sonido

Existen 6 comandos de sonido en la SPU de Vircon32. Por un lado podemos operar sobre un canal determinado, con comandos para reproducir, pausar o detener. Y por otro la SPU nos permite detener, pausar o reanudar todos los canales a la vez.

Tenemos más de una opción para reproducir un sonido desde C. Podemos simplemente usar `play_sound()` y dejar que busque un canal libre por nosotros. Aunque en algunos casos (por ejemplo, para música de fondo) nos puede interesar elegir un canal determinado para reproducir. En código, las 2 maneras se usan así:

```
// reservamos el canal 15 para la musica
play_sound_in_channel( Level1Music, 15 );

// reproducimos un sonido en cualquier canal
// (pero necesitamos saber cual para cambiar el volumen)
int UsedChannel = play_sound( Level1Music );
set_channel_volume( UsedChannel, 0.25 );
```

Los comandos para todos los canales pueden sernos útiles en situaciones en las que un juego cambia de contexto. Por ejemplo podemos pausar todos los sonidos en un menú de pausa, y reanudarlos al salir de él de esta forma:

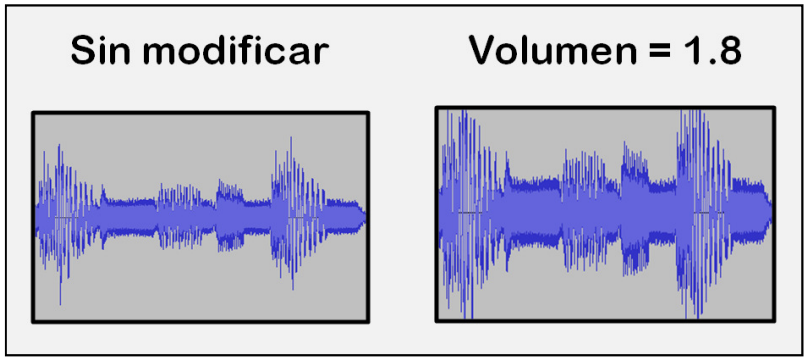
```
// al entrar en el menu de pausa
pause_all_channels();

// mostrar el propio menu
ShowPauseMenu();

// al volver al juego
resume_all_channels();
```

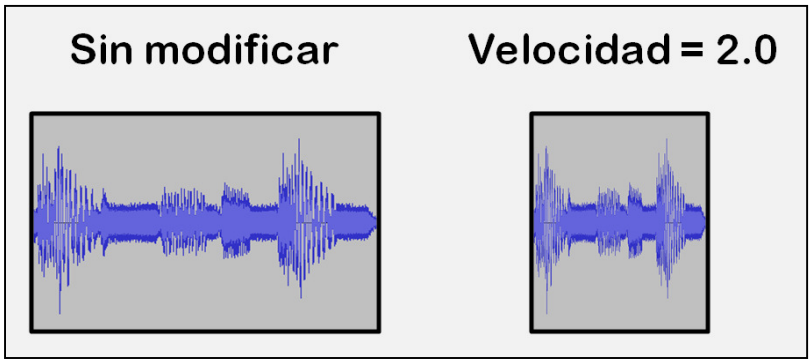
Volúmen de reproducción

En los canales de reproducción el volúmen se puede modificar con `set_channel_volume()`, en un rango de 0 a 10. El volúmen por defecto es 0.5. El volúmen de un canal no cambia a no ser que lo cambiemos nosotros, y será el mismo aún si reproduce distintos sonidos.



Velocidad de reproducción

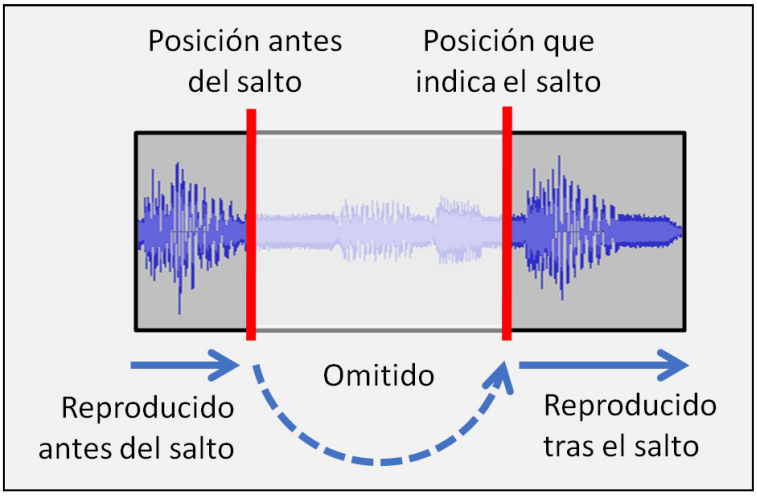
Los canales también permiten modificar la velocidad de reproducción, usando la función `set_channel_speed()`. Si se reproduce a diferente velocidad, también cambia el tono del sonido. El rango es de 0 a 100, y la velocidad por defecto es 1. Este efecto puede ser útil para simular distintas notas de un instrumento.



Al igual que con el volúmen, la velocidad de un canal se mantendrá si no la cambiamos.

Saltos en la reproducción

Los canales reproducen su sonido asignado avanzando continuamente su posición de reproducción a lo largo de los samples de ese sonido. Los canales permiten modificar esa posición de reproducción usando la función `set_channel_position()`, dando lugar a saltos. Por ejemplo, en el caso de un salto hacia delante, la reproducción de un sonido sucedería como se muestra en la imagen de debajo.

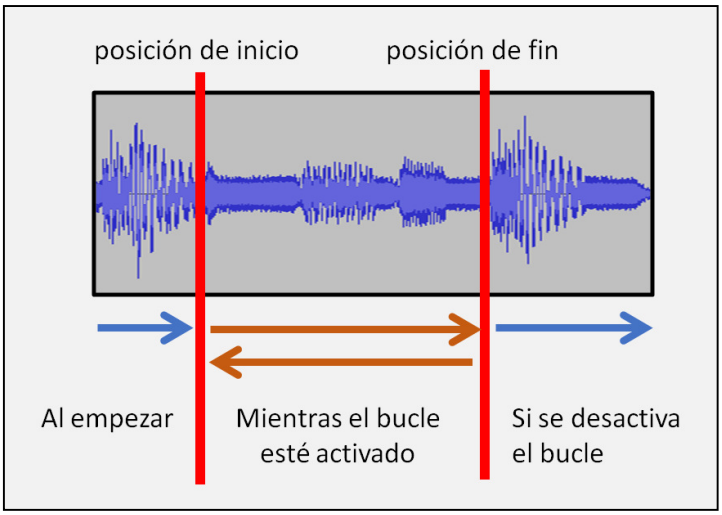


Reproducción en bucle

Los canales de la SPU nos permiten reproducir sonidos en bucle. Esta opción se puede activar a nivel de canal, con `set_channel_loop()`, pero también cada uno de los propios sonidos de la GPU tiene un parámetro para determinar si se debe reproducir en bucle.

Esto se controla con `set_sound_loop()`. Cada vez que un canal comienza a reproducir cualquier sonido, el canal automáticamente activará o desactivará la reproducción en bucle para adaptarse a ese sonido concreto. Por defecto todos los sonidos y canales tienen el bucle desactivado.

Por defecto el bucle se hace para el sonido completo, pero podemos hacer bucles parciales marcando puntos de inicio y fin. La reproducción será entonces como en esta imagen:



Para conseguir esto necesitaremos delimitar en ese sonido la región concreta que debe reproducirse en bucle. Lo podemos hacer como en el siguiente ejemplo:

```
// configurar este sonido para que se reproduzca en bucle
select_sound( SoundCarEngine );
set_sound_loop( true );

// configurar la region de bucle (posiciones dadas en samples)
set_sound_loop_start( 1348 );
set_sound_loop( 12507 );

// al reproducir se aplica el bucle automaticamente
int EngineChannel = play_sound( SoundCarEngine );

// (...)

// desactivar el bucle para dejar que el sonido termine
select_channel( EngineChannel );
set_channel_loop( false );
```

Memoria RAM

La memoria RAM es un componente pasivo, con lo cual no hay mucho que saber sobre ella. Su tamaño es de 16 MB, aunque como la unidad mínima es la palabra de 32 bits (word), es más adecuado medir el tamaño como 4 MW.

La BIOS

La BIOS es un software interno que siempre existe en la consola. Cuando la consola arranca nos mostrará el logo de Vircon32, y después comprueba que haya un cartucho conectado. Si lo hay, le transferirá el control y si no mostrará una pantalla de aviso.

Si en algún momento sucede un error hardware, la BIOS recibirá el control de nuevo para gestionarlo y al terminar detendrá la ejecución.

Una BIOS de Vircon32 siempre tiene exactamente 1 textura y 1 sonido, que tienen reservadas las IDs -1 en GPU y SPU. Uno de los elementos que incluye esa textura es una fuente de texto de 10x20 pixels para permitir escribir texto en la pantalla.

Controlador de cartucho

Este controlador permite a la consola acceder a los contenidos almacenados en el cartucho (programa, texturas y sonidos). Todo esto sucede automáticamente, y no necesitamos hacer nada para ello en nuestro programa.

Además también se le puede pedir al controlador informaciones básicas sobre el cartucho que hay en la consola: si está conectado, cuántas texturas y sonidos incluye, y el tamaño de la ROM de programa. Estas informaciones normalmente no se necesitan desde un juego, que ya sabe los elementos que utiliza. Así pues, sólo tendría sentido pedir informaciones al controlador de cartucho desde programas de test.

La BIOS sí necesita consultar si hay un cartucho conectado durante el arranque. Después ya no son necesarias más comprobaciones: en Vircon32, cuando la consola se enciende la entrada de cartucho se bloquea y ya no se puede introducir ni extraer ningún cartucho sin apagar primero la consola.

Controlador de tarjeta de memoria

La tarjeta de memoria, cuando está conectada, es accesible a la CPU como una región de memoria de lectura/escritura. Su tamaño es de 1 MB, aunque como la unidad mínima es la palabra de 32 bits (word), es más adecuado medir el tamaño como 256 KW. En C podemos comprobar si hay una tarjeta en la consola con `card_is_connected()`. Esto es importante, ya que tratar de acceder a la tarjeta causará un error hardware si no hay ninguna conectada.

La firma de cada juego

Un programa puede acceder a la tarjeta como si fuera una zona de memoria cualquiera, pero ya que puede contener partidas guardadas (incluso de otros juegos), se recomienda asegurarse de lo que contiene antes de intentar usar la tarjeta.

Para esto la práctica habitual es que las primeras 20 palabras de la tarjeta se usen como "firma" del juego, es decir, un conjunto de valores conocidos que identifique si es nuestro juego el que guardó los datos (si la firma coincide con la esperada) o fue otro. Así, si al comprobar la tarjeta nuestro juego detecta una firma desconocida, evitamos que trate de cargar una partida que será incompatible y puede dar errores.

Podemos trabajar con firmas usando `card_read_signature()`, `card_write_signature()` y `card_signature_matches()`, de la forma que mostramos en este ejemplo:

```
// creamos una firma para nuestro juego
game_signature GameSignature;
memset( GameSignature, 0, sizeof(game_signature) );
strcpy( GameSignature, "MY TEST GAME" );

// (...)

// al guardar la partida
card_write_signature( &GameSignature );
SaveGameData();


// (...)

// al cargar la partida
if( card_signature_matches( &GameSignature ) )
  LoadGameData();
```

Una tarjeta también puede estar vacía, si aún no se ha usado tras ser creada. Un programa puede suponer que una tarjeta está vacía si la firma sólo contiene ceros. En la librería estándar contamos con `card_is_empty()` para comprobarlo.

Una vez que hemos comprobado que podemos usar la tarjeta, podemos leer y escribir datos en ella usando `card_read_data()` y `card_write_data()`. Por ejemplo, las funciones de cargar y guardar del ejemplo anterior podrían tener esta forma:

```
// los datos de nuestro juego
game_signature MySignature;    // tipo ya definido en "memcard.h"
GameState CurrentState;        // estructura definida por nosotros

// (más adelante en nuestro programa)
void SaveGameData()
{
    // los datos van a continuacion de la firma
    int OffsetInCard = sizeof( game_signature );
    card_write_data( &CurrentState, OffsetInCard, sizeof( CurrentState ) );
}
```

```
void LoadGameData()
{
    // los datos van a continuacion de la firma
    int OffsetInCard = sizeof( game_signature );
    card_read_data( &CurrentState, OffsetInCard, sizeof( CurrentState ) );
}
```

Generador de números aleatorios

Este generador parte de un número inicial (una "semilla"), y a partir de él es capaz de producir una secuencia de números pseudoaleatorios cada vez que la CPU se lo pide. A partir de una misma semilla el generador crea siempre la misma secuencia, incluso en consolas Vircon32 distintas.

Podemos cambiar la semilla con `srand()`, y extraer números de la secuencia con llamadas repetidas a `rand()`. En programas en C es habitual crear una semilla usando la hora actual para hacer que en cada partida la secuencia generada sea diferente.

```
// iniciamos una nueva secuencia de numeros
srand( get_time() );

// simulamos la tirada de un dado: de 1 a 6
int DiceThrow = 1 + rand() % 6;
```