

# **Desarrollo de Videojuegos**

## **Conceptos Básicos para Desarrollo de Videojuegos 2D**

Roberto Albornoz Figueroa © 2006-2007  
[ralbornoz@gmail.com](mailto:ralbornoz@gmail.com)  
<http://www.blogrcaf.com>

## Licencia Creative Commons

### Reconocimiento-NoComercial-SinObraDerivada 2.5



**Usted es libre de:**

- copiar, distribuir y comunicar públicamente la obra.

**Bajo las condiciones siguientes:**



**Reconocimiento.** Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciadador.



**No comercial.** No puede utilizar esta obra para fines comerciales.



**Sin obras derivadas.** No se puede alterar, transformar o generar una obra derivada a partir de esta obra.

- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor

## Contenidos

<b>Introducción</b>	<b>5</b>
<b>Conceptos básicos</b>	<b>6</b>
Monitor	6
Tarjeta de video	6
Frecuencia de refrescado	6
Vertical retrace	6
Pixel	7
Bitmap	7
Profundidad de Color (BPP)	7
Cantidad de colores	7
Modelo RGB	8
Paleta	8
Modo indexado	8
Cantidad de bits en un pixel	9
Resolución	9
Modo de video	9
Video RAM	10
<b>Técnicas 2D</b>	<b>12</b>
Sprites	12
Animación de Sprites	12
Transparencias	14
Color keys	14
Máscara	15
Transformaciones	16
Translation	16
Rotation	16
Scaling	17
Flipping	18
Alpha blending	18
Screen buffer	19
Surface	21
Blitting	21
Double Buffering	22

<b>Dirty Rectangles</b>	<b>24</b>
<b>Clipping</b>	<b>24</b>
<b>Sistema de coordenadas 2D</b>	<b>24</b>
<b>Sincronización en los videojuegos</b>	<b>26</b>
<b>Sincronización por Framerate</b>	<b>26</b>
<b>Sincronización por Tiempo</b>	<b>27</b>
<b>Resumen</b>	<b>29</b>

---

## **Introducción**

Una de las razones para comenzar a desarrollar primero videojuegos 2D, es que los conceptos involucrados son mucho más simples y fáciles de asimilar.

Además lo más probable es que varios de estos conceptos ya sean conocidos por quienes lleven un tiempo programando o trabajando con computadores.

Otra ventaja, es que obtendremos resultados más rápidos, a diferencia del desarrollo de videojuegos 3D, que involucra conocer tópicos más avanzados de matemáticas y de programación. Pero tampoco hay que asustarse, si queremos luego dar el próximo paso a las 3D, ya que hay que recordar siempre que todo lo podemos estudiar y aprender, nada es imposible.

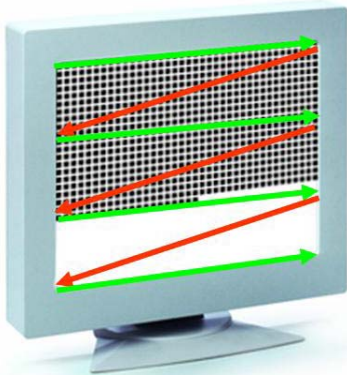
En las próximas páginas conoceremos la terminología básica, que nos hará comprender de mejor forma los conceptos gráficos involucrados en un videojuego.

## Conceptos básicos



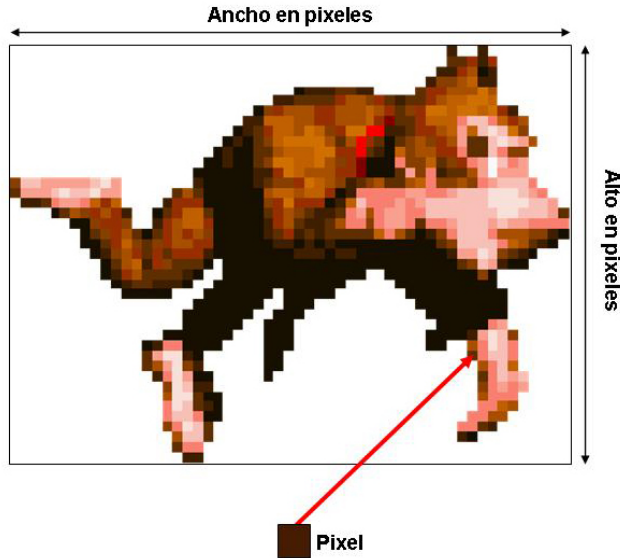
Partiremos hablando del dispositivo externo más común y que todos ya conocemos, el **Monitor**. Este dispositivo de salida es uno de los más importantes que posee nuestro computador, ya que nos permite *visualizar* la información con la que estamos trabajando.

El monitor se conecta al computador a través de una **Tarjeta de video**. Esta tarjeta es un circuito integrado que nos permite transformar las señales digitales con las que trabaja, en señales análogas para que puedan ser visualizadas en el monitor. Este proceso es realizado a través de un conversor analógico-digital que toma la información que se encuentra en la memoria de video y luego la lleva al monitor. Este conversor se conoce como **RAMDAC** (Random Access Memory Digital Analog Converter).



La superficie de la pantalla de un monitor es fluorescente y esta compuesta por líneas horizontales. El hardware del monitor actualiza de arriba hacia abajo cada una de estas líneas. Para dibujar una imagen completa el monitor lo hace a una cierta velocidad, a esto se le conoce como **Frecuencia de refresco** y se mide en Hz. Los valores frecuentes son de 60/70 Hz, incluso frecuencias mayores. Es preferible que supere los 70 Hz para que la vista no se canse tanto y no aparezcan los molestos parpadeos (flickering). El valor de la frecuencia depende de la resolución usada.

El tiempo que existe entre dos refrescos de pantalla se conoce como **Vertical retrace**, y corresponde al momento en que el haz de electrones del monitor regresa a la parte superior de la pantalla.



El concepto más básico usado en la programación gráfica es el de **Pixel**, significa Picture Element, es decir, un elemento de una imagen que corresponde a la unidad mínima que esta puede contener. También lo podríamos definir como un simple punto que es parte de una imagen determinada, que podemos mostrar en pantalla y que tiene asociado un color.

Esta imagen que está formada por un arreglo ordenado de pixeles en forma de grilla o cuadrícula, se conoce con el nombre de **Bitmap**. Un bitmap posee

dimensiones: *ancho* y *alto* (que se miden en número de píxeles) y además tiene asociado un formato, que puede ser alguno de los ya conocidos: bmp, png, jpg, gif, etc.

La diferencia entre formatos gráficos, viene dada por algunos parámetros que esperamos tener, como la calidad o tamaño del archivo. Por ejemplo si queremos mantener un equilibrio en tamaño/calidad, un formato que se ajusta a estos parámetros son los formatos jpg o png, no así el formato bmp, que casi no posee ningún tipo de compresión.

Una imagen tiene otras características, aparte de sus dimensiones, una de estas es la **Profundidad de color**. Esto corresponde al número de bits necesarios para poder representar un color, es conocido por las siglas **BPP** (Bits Per Pixel).

El número de bits que podemos utilizar es de 1, 8, 16, 24 o 32. Con este dato podemos saber cual será el número máximo de colores que puede representar una imagen en pantalla. El número de combinaciones se calcula mediante la siguiente fórmula:

$$\text{Cantidad de Colores} = 2^{\text{bpp}}$$

Ahora podemos resumir en una tabla cada una de estas combinaciones:

Número de bits	Cantidad de colores	Nombre
1	2	Monocromo
8	256	Indexado
16	65.536	Color de alta densidad (High Color)
24	16.777.216	Color real (True Color)
32	4.294.967.296	Color real (True Color)

El color se representa en pantalla utilizando el **Modelo RGB** (Red, Green, Blue), es decir, para formar un color necesitamos conocer la intensidad de estos colores primarios. Cada una de estas intensidades corresponde a un número que se encuentra en el rango: *0 a 255*.

Cuando se utilizan 8 bits para mostrar un pixel, como ya sabemos, tenemos como máximo 256 colores, pero no se representan directamente utilizando las intensidades RGB, sino que se mantiene una **Paleta** o **Colormap** (un arreglo de tamaño igual a 256 bytes), donde cada posición apunta a un color con las 3 componentes (Rojo, Verde, Azul). Esta paleta puede ser modificada para implementar algunos efectos, ya que si cambiamos por ejemplo las componentes de un color, se verán modificados inmediatamente todos los pixeles que hagan referencia a dicho color. El modo de 8 bits, se conoce también como **Modo indexado** y ofrece un alto rendimiento, pero como ya sabemos con la desventaja de tener pocos colores disponibles.

Para profundidades más altas, esta paleta ya no existe, y esto nos permite tener un abanico más grande de colores. En la actualidad ya no usamos modos de video a 8 bits, normalmente usaremos 16 o 32, siendo el modo de 32 bits el más eficiente, ya que los procesadores de 32 bits trabajan con datos empaquetados en bloques de 4 bytes.

Todas las componentes que forman un color se empaquetan en un entero, donde se asigna una cantidad fija de bits para cada una de ellas.

El orden en que se empaquetan las componentes de un color, depende de la arquitectura en la que se esté trabajando, podemos usar el formato RGB (Big Endian) o BGR (Little Endian).

En arquitecturas x86 (Windows, Linux) se usa el formato *Little Endian* (el byte de menor peso se almacena en la dirección más baja de memoria y el byte de mayor peso en la más alta) y en PowerPC (Mac), *Big Endian* (el byte de mayor peso se almacena en la dirección más baja de memoria y el byte de menor peso en la dirección más alta).

La **cantidad de bits** usada para cada componente de un pixel, la podemos ver resumida en la siguiente tabla:

8 bits	16 bits	24 bits	32 bits
Se utiliza una paleta, donde cada posición apunta a un color que posee las intensidades RGB.	<p><b>1:5:5:5</b></p> <p>El primer bit indica la transparencia, si es 0, la transparencia será total, si es 1 habrá opacidad total.</p> <p>5 bits para cada componente RGB.</p>	8 bits para cada componente RGB	<p><b>8:8:8:8</b></p> <p>El primer byte corresponde al canal alpha (componente que almacena el grado de transparencia)</p> <p>Los últimos 3 bytes son las componentes RGB del color.</p>
	<p><b>5:6:5</b></p> <p>5 bits para componente Red 6 bits para componente Green 5 bits para componente Blue</p>		

Anteriormente mencionamos que una imagen tiene dimensiones y a esta dimensión (ancho y alto) también se le conoce como **Resolución**. La resolución nos permite conocer cuanto detalle existe en una imagen, esto quiere decir que a mayor resolución obtendremos mayor calidad. La resolución no solo se aplica a una imagen, sino que también a la pantalla o monitor. Existen algunas resoluciones estándar:

- 320x200
- 320x240
- 640x480
- 800x600
- 1024x768
- 1152x864
- 1280x720
- Etc.

El **Modo de video** es la unión de la *resolución* con la *profundidad de color* y se denota como **AnchoxAltoxBpp**.

Por ejemplo podríamos trabajar con los siguientes modos de video:

- 640x480x8 (640 pixeles de ancho x 480 pixeles de alto a 8 bits por pixel)
- 640x480x16 (640 pixeles de ancho x 480 pixeles de alto a 16 bits por pixel)
- 800x600x32 (800 pixeles de ancho x 600 pixeles de alto a 32 bits por pixel)
- Etc.

La elección de un modo de video u otro, es un factor importante a la hora de comenzar a desarrollar un videojuego, ya que con estos datos sabremos cual será el flujo de información (medido en bytes) que existirá. Además las imágenes que se muestren en pantalla deberán tener algún tamaño determinado, que debe ajustarse a la resolución elegida.

Por ejemplo supongamos que estamos trabajando con el siguiente modo de video, 800x600x32 y queremos conocer cuando memoria ocupa una imagen con las mismas dimensiones que la resolución seleccionada, entonces el tamaño que ocupará será de:

$$\text{Tamaño} = 800 \times 600 \times 4 = 1.920.000 \text{ bytes} = 1.83 \text{ Mb}$$

Multiplicamos por 4, ya que  $32/8 = 4$  bytes, es decir para mostrar un solo color en pantalla o píxel necesitamos 4 bytes.

El resultado anterior no es un tamaño despreciable, imaginemos el tamaño que podríamos obtener a resoluciones mayores.

En los primeros juegos que aparecieron para PC, se utilizaban resoluciones muy bajas como 320x200 (también conocido como modo 13h) o 320x240 (modo x) y a 8 bits por píxel, o sea como ya sabemos disponíamos de un máximo de 256 colores.

A medida que pasaron los años se empezaron a utilizar resoluciones de 640x480 o 800x600 y a profundidades de color más altas, de 16 o 32 bits, todo esto gracias a que aumentaron las capacidades de las tarjetas de video.

Hoy en día, en la mayoría de los juegos tenemos la posibilidad de elegir el modo de video que mejor se adapte a nuestro equipo. Existen algunos casos típicos, como cuando queremos tener el mejor rendimiento en nuestro videojuego, bajamos la resolución al mínimo o a una resolución intermedia. Al contrario, cuando disponemos de un buen equipo y una buena tarjeta de video, lo más probable es que seleccionemos un modo de video más alto, para disfrutar de unos gráficos de mejor calidad.

Otro concepto importantísimo es el de **Video RAM**, básicamente es la memoria o buffer que se encuentra en la tarjeta de video. Toda tarjeta de video tiene como mínimo una memoria, un microprocesador, una caché y un bus de datos, que normalmente es AGP.

En la Video RAM se almacenan los datos que serán mostrados posteriormente en la pantalla, estos datos pueden ser texturas, vértices, etc.

Para el desarrollo de videojuegos 2D, esto se simplifica ya que solo tendremos en la memoria cada uno de los píxeles que forman la imagen.

En el caso de juegos 3D, en la memoria se pueden almacenar algunas rutinas o procedimientos que ejecutan algunas instrucciones para generar normalmente algún efecto, a estas rutinas se les conoce como Shaders.

Además, el microprocesador también conocido como GPU (Graphics Processing Unit), se comunica con la memoria a través de un bus interno, ejecuta instrucciones y actualiza el caché de instrucciones, de texturas y geometría. El proceso de llenar la memoria de video con datos se hace posible a la existencia de la GPU, el cual lo hace a través de un bus de puerto acelerado, conocido como AGP. Después el monitor lee periódicamente desde el buffer de video (Video RAM) los datos y refresca la pantalla, lo que nos permite ver una imagen en el monitor. No importa la velocidad con que se modifiquen los datos del buffer, el monitor no actualizará nada hasta el próximo refresco.

## Técnicas 2D

Ahora continuaremos viendo otros conceptos, que llamaremos técnicas 2D. Estos términos son aplicados básicamente al desarrollo de aplicaciones 2D y algunos son la base para las 3D.

### Sprites



Un sprite es cualquier objeto que aparece en nuestro videojuego. Normalmente tiene asociado algunos atributos, siendo los más básicos una imagen y una posición.

Los sprites pueden ser *estáticos* o *dinámicos*. Al hablar de *sprite estático*, nos referimos a un objeto que no cambiará su posición durante el videojuego, por ejemplo la imagen de una llama, piedra, etc. En cambio un *sprite dinámico*, es aquel que tendrá algún tipo de movimiento, el cual puede ser realizado por el usuario o por el computador, por ejemplo el sprite de un jugador o de los enemigos.

Representar un sprite en memoria puede ser realizado de muchísimas formas, y puede depender mucho del gusto del programador y su experiencia. Normalmente se almacenan sus atributos en una estructura o clase.

### Animación de Sprites

Esto es bastante simple, aquí aparece el concepto de **frame** o cuadro, que corresponde a una de las imágenes que forman la animación completa de un sprite. Podemos pensar en una secuencia de imágenes que son dibujadas rápidamente, que engañan al ojo humano dando el efecto de un movimiento fluido. Es el mismo método que se utiliza para los dibujos animados.



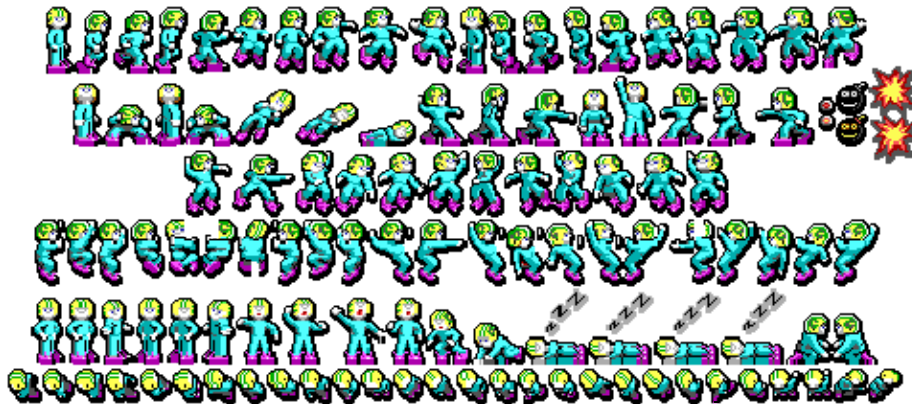
El número de frames dibujados en un periodo de tiempo se conoce como **frame rate** (tasa de frames), siendo un aspecto muy importante en la animación, ya que de esto depende la calidad de animación que obtendremos. Si cada uno de los frames se muestra muy rápido o muy lento, la ilusión del movimiento se perderá y el usuario verá cada uno de los frames como una imagen separada.

Una animación muy común puede ser la de un personaje caminando, saltando, o realizando cualquier otra acción.

Esto también debe ser representado de alguna forma inteligente en nuestro videojuego, utilizando algún tipo de estructura de datos adecuada, como un vector o lista.

Normalmente el sprite tendrá como atributo una lista de animaciones (las acciones del personaje, enemigo, etc.), donde cada animación corresponderá a otra lista con cada uno de los frames.

La totalidad de los frames que forman un personaje, suele almacenarse en un solo archivo de imagen, a esto se le llama **sprite sheet**, es decir, una imagen donde tenemos secuencias de frames para las diferentes acciones que puede realizar un personaje en un videojuego. La siguiente imagen lo ilustra claramente.

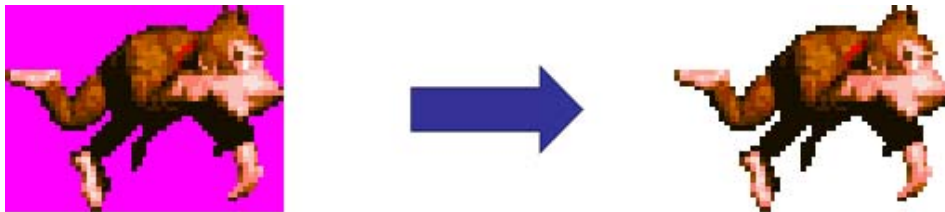


## Transparencias

### Color keys

Cuando mostramos una imagen en pantalla, esta siempre se verá como un cuadrado o rectángulo, pero sabemos que dentro tiene la representación de un objeto, personaje, enemigo, ítem, etc. Al dibujar esta imagen queremos ver solo la forma que representa, para esto debemos elegir algún color transparente, pintar con ese color las zonas que no queremos que aparezcan y luego dibujar la imagen ignorando dicho color.

Normalmente se utiliza el color magenta (255, 0, 255), como color de transparencia, ya que es poco probable que se encuentre en una imagen.



Dependiendo de la API gráfica que utilicemos para desarrollar nuestro videojuego, tendremos una función que realice esta tarea, la de descartar un color determinado al dibujar la imagen.

Pero no solo usar un color como el magenta es la solución para no ver una imagen en su forma original, también podemos hacer uso de las propiedades del formato gráfico PNG (Portable Network Graphics), ya que este formato guarda un canal alpha con las partes transparentes de la imagen.

## **Máscara**

Existe otro método para ignorar un color en una imagen, el uso de una máscara que se antepone al bitmap original, es decir, debemos tener otro bitmap que indica cuales son los pixeles transparentes. El color que indica la transparencia es el negro, y el color blanco la figura que queremos mostrar. Por lo tanto, al dibujar la imagen, recorreremos cada pixel de la máscara, y si este color es blanco, mostraremos el pixel del bitmap original que se encuentra en esa posición.



## Transformaciones

Cuando dibujamos un sprite en pantalla, tenemos la posibilidad de aplicar algunas transformaciones, las más usadas son las siguientes:

### Translation

Esta es la transformación más simple, corresponde a cambiar la posición del sprite para luego dibujarla en otro lugar de la pantalla, dando el efecto de movimiento, sin duda lo más usado en cualquier videojuego. Y se resume en asignar una nueva posición a las coordenadas (x, y) del sprite.

Por ejemplo si las variables **x** e **y** son las coordenadas de la esquina superior izquierda del sprite, y **vx**, **vy** las velocidades en cada eje, con una simple suma en cada componente cambiaremos la posición del sprite.

$$x = x + vx;$$

$$y = y + vy;$$

### Rotation

Otra transformación típica, que consiste en girar el sprite un número determinado de grados. Se usa para mostrar objetos vistos desde otro ángulo, por ejemplo en el videojuego Micro Machines, cuando doblamos cambiamos el ángulo del auto y luego este cambio lo vemos reflejado en la pantalla gracias a la rotación.

También lo podríamos usar para realizar efectos para una presentación, efectos durante el videojuego, etc. Además al rotar una imagen debemos saber con respecto a que punto lo haremos (pivote), por ejemplo en la siguiente imagen realizamos unas rotaciones con respecto al centro de la imagen.

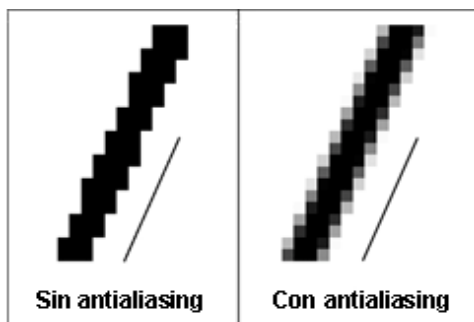
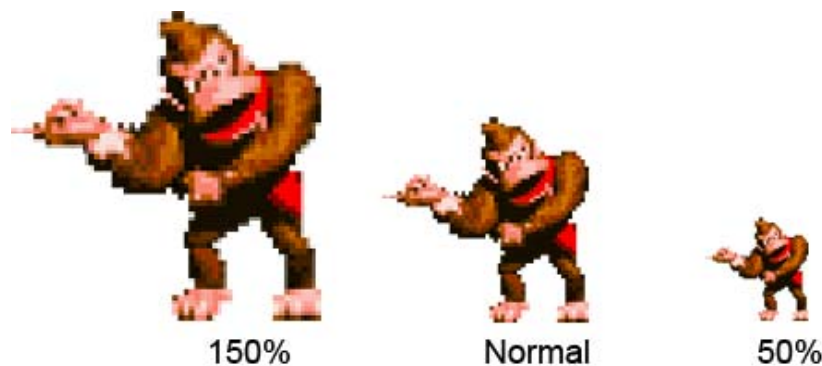


## Scaling

Otra transformación muy usada en algunos videojuegos, consiste en escalar una imagen, es decir, cambiar su tamaño (normalmente de forma proporcional), ya sea un aumento o una disminución. Se puede usar para dar un efecto de profundidad, es decir, si el objeto está más alejado de nosotros lo dibujaremos más pequeño, en cambio si está muy cerca de nosotros, lo dibujaremos de un tamaño más grande.

Suele usarse también en conjunto con la rotación para realizar algunos efectos.

Al utilizar esta transformación, hablamos de **scale factor**, es decir, un valor que indica el porcentaje de escalado, donde el valor 1 corresponde al tamaño normal.



En la imagen anterior nos damos cuenta que al aumentar su tamaño, se produce un efecto no deseado en los bordes de la imagen, conocido como **aliasing**, es decir, las líneas o curvas que forman los bordes de la imagen se ven discontinuas (en forma de escalera). Para esto existe una solución, el **antialiasing**, una técnica que permite suavizar todos los bordes y así disminuir el

efecto de escalera. En toda API gráfica encontraremos una función que realice esta tarea.

## Flipping

El flipping es un tipo de transformación especial para realizar algunos efectos.

Básicamente existen dos tipos, *Vertical Flip* que corresponde al efecto que se produce cuando se refleja un objeto en el agua y *Horizontal Flip* que corresponde al reflejo de un objeto en un espejo. Con la siguiente imagen quedará mucho más claro.



## Alpha blending

Alpha blending o mezclado alpha es una técnica para crear transparencias en una imagen con respecto al fondo. Para lograr esto se agrega un cuarto canal a un color, llamado **canal alpha**.

Ahora los colores de una imagen se representarán como RGBA. Cada valor alfa de un pixel representa su grado de opacidad, donde un valor cero indica un 100% de transparencia, y a medida que aumentamos su valor se irá haciendo más opaco. El rango va desde 0 a 255.

Para crear un canal alpha en una imagen, tenemos dos opciones, una es almacenar esta información con algún software de edición de imágenes (Photoshop) y en un formato apropiado. (PNG, Portable Network Graphics), o hacerlo posteriormente en el mismo videojuego, utilizando alguna función de la API que nos permita setear un grado de transparencia para la imagen.

El funcionamiento de esta técnica es relativamente simple, ya que se realiza una media ponderada de cada pixel que forma la imagen y los pixeles que conforman el fondo.

Por ejemplo supongamos que tenemos un color de una imagen, el **(100, 40, 50)** con un valor alfa de **90**, y dibujaremos este pixel sobre otro con el valor **(70, 100, 80)**. Con estos datos el pixel resultante tomara un **35%** del primer color y un **65%** del segundo.

Entonces, multiplicaremos cada componente del pixel por el porcentaje que le corresponde y luego sumaremos cada unas de las componentes, obteniendo así el pixel resultante.

**Pixel1:** (100, 40, 50) / \*0.35

**Pixel2:** (70, 100, 80) / \*0.65

**Pixel resultante:**  $(100*0.35+70*0.65, 40*0.35+100*0.65, 50*0.35+80*0.65) =$   
**(81, 79, 70)**

El cálculo anterior se debe hacer para cada pixel que forma la imagen, lo cual afecta al rendimiento, pero hoy en día todas las tarjetas de video realizan este tipo de operaciones vía hardware.



Esta técnica se puede aplicar a un bitmap completo o solo a un grupo de pixeles de la imagen. Es utilizado para hacer una variedad de efectos en videojuegos, por ejemplo puede usarse para dar un carácter de fantasma a cierto personaje, para hacer un menú transparente, escribir texto sobre un rectángulo transparente, etc.

### Screen buffer

El screen buffer es simplemente un área de la memoria de video, donde podemos dibujar algo que se mostrará en la pantalla. Si queremos mostrar un pixel, debemos tener un puntero a la dirección de memoria de video y calcular la posición (offset, desplazamiento) donde queremos dibujar el pixel y luego en esa posición copiar el byte o los bytes que representan el color.

Supongamos que estamos trabajando en el siguiente modo de video: 640x480x24, es decir, tenemos 640 pixeles de ancho, 480 pixeles de alto, y cada pixel ocupa 3 bytes, entonces:

```
int color; // Acá almacenamos el color del pixel
int *buffer; // Screen Buffer
int x, y; // Coordenadas del punto

// Calculamos el desplazamiento
int desp=y*640 + x*3;

// Colocamos un pixel en la posición (x,y)
buffer[desp]=color;
```

En el ejemplo anterior no mostramos como se calcula el valor numérico del color, para hacerlo hay que hacer uso de operaciones a nivel de bits y desplazamientos, y esto depende de la profundidad de bits que se este usando. Supongamos que el color tiene la siguientes componentes **(200, 150, 60)** y estamos trabajando a **24 bits**. Si recordamos la tabla de componentes RGB, para 24 bits tenemos que cada componente ocupa 8 bits, así que el cálculo es el siguiente:

```
int r=200;
int g=150;
int b=60;

Color = (r << 16) | (g << 8) | b;
```

Es decir, gracias a los desplazamientos colocamos la componente en su posición correcta y luego sumamos todos estos valores con el operador OR a nivel de bits.

Para calcular la posición de memoria donde debe colocarse el pixel, la posición “**y**” es multiplicada por el **ancho** en pixeles de la resolución elegida, es decir, con esto nos movemos “**y**” líneas por la pantalla, luego en esa posición debemos movernos unos ciertos bytes hasta llegar a la posición “**x**” que buscamos. Para eso sumamos la posición de la coordenada “**x**” multiplicada por el tamaño del pixel en bytes, en este caso como estamos usando una profundidad de color de 24 bits, se debe multiplicar por 3. Luego asignamos el color al buffer justo en la posición que calculamos anteriormente.

En general esta es la forma elemental para dibujar un pixel en pantalla. Para dibujar una imagen completa debemos recorrer cada punto de la imagen, calcular la posición y asignar el color a dicha posición. Para nuestra suerte, todas las APIs gráficas traen funciones para crear un color, como lo hicimos anteriormente, y funciones para dibujar una imagen completa en pantalla.

## **Surface**

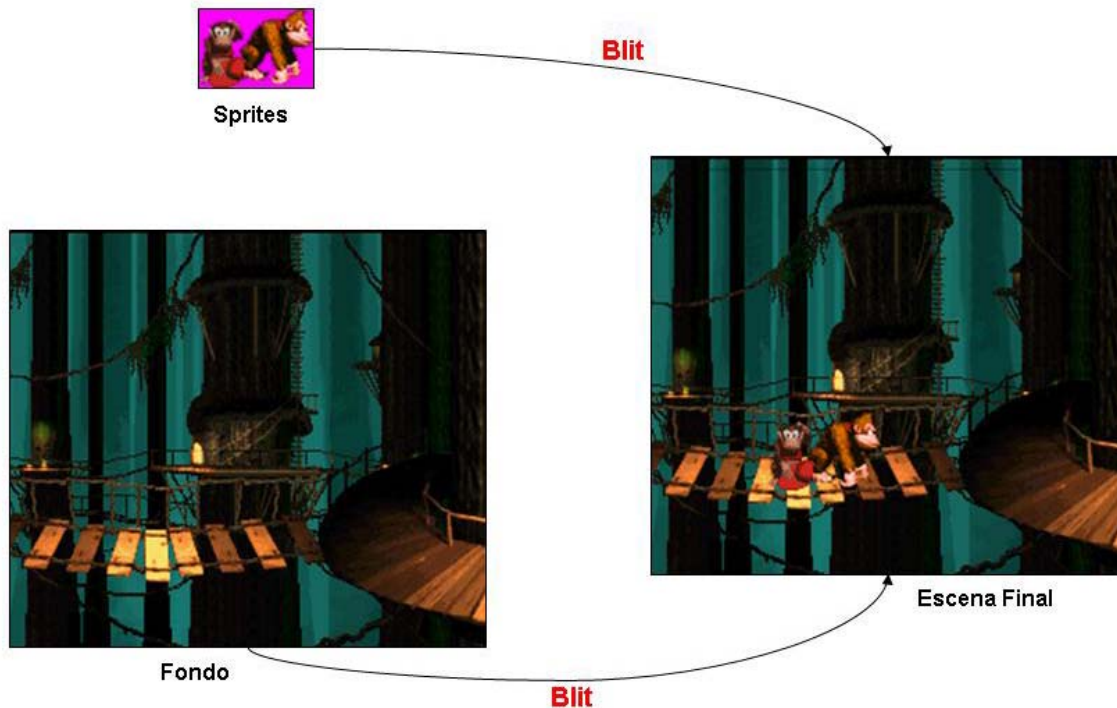
Este término es la base de la programación 2D. Básicamente es una zona de memoria que puede estar en la RAM o Video RAM y es usada para almacenar un bitmap. Si queremos ver esto en el ámbito de la programación, una **Surface** es una estructura o clase que tiene como mínimo las mismas propiedades de un archivo de imagen, es decir, tendrá un arreglo lineal de datos que contiene cada uno de los pixeles que forma la imagen y además el ancho y alto del bitmap. Dependiendo de la biblioteca gráfica que se use, pueden existir más atributos.

Una Surface se podría confundir con el concepto de Sprite, pero en general un Sprite tiene como atributo al menos una Surface.

## **Blitting**

El blitting es una operación para transferir un bloque de bytes (surface) de un sector de la memoria a otra. Viene del termino **Blit** (Block Image Transfer, Transferencia de Imagen por Bloque) y es una forma de renderizar (dibujar) sprites con o sin transparencias sobre un fondo. Esta técnica puede ser acelerada por hardware, lo cual ayuda a incrementar bastante el rendimiento, de hecho esta operación es una de las más críticas en cualquier videojuego y es la que más usaremos.

Gracias al blitting podemos armar una escena en un videojuego, por ejemplo en la siguiente imagen vemos que a partir de dos superficies, una con los sprites de los personajes y otra con un fondo, armamos a través de blits una tercera superficie que contiene la escena final.



En una operación de blitting, podemos especificar en que posición de la superficie destino queremos copiar la superficie origen, incluso que parte de la superficie origen queremos copiar. De esta forma podemos colocar en cualquier posición, superficies dentro de otras. La forma de especificar las superficies origen/destino y sus posiciones, depende de la API gráfica que se esté utilizando. Siempre se nos proveerá de una función de blitting, ya que todas las bibliotecas disponen de al menos una.

### **Double Buffering**

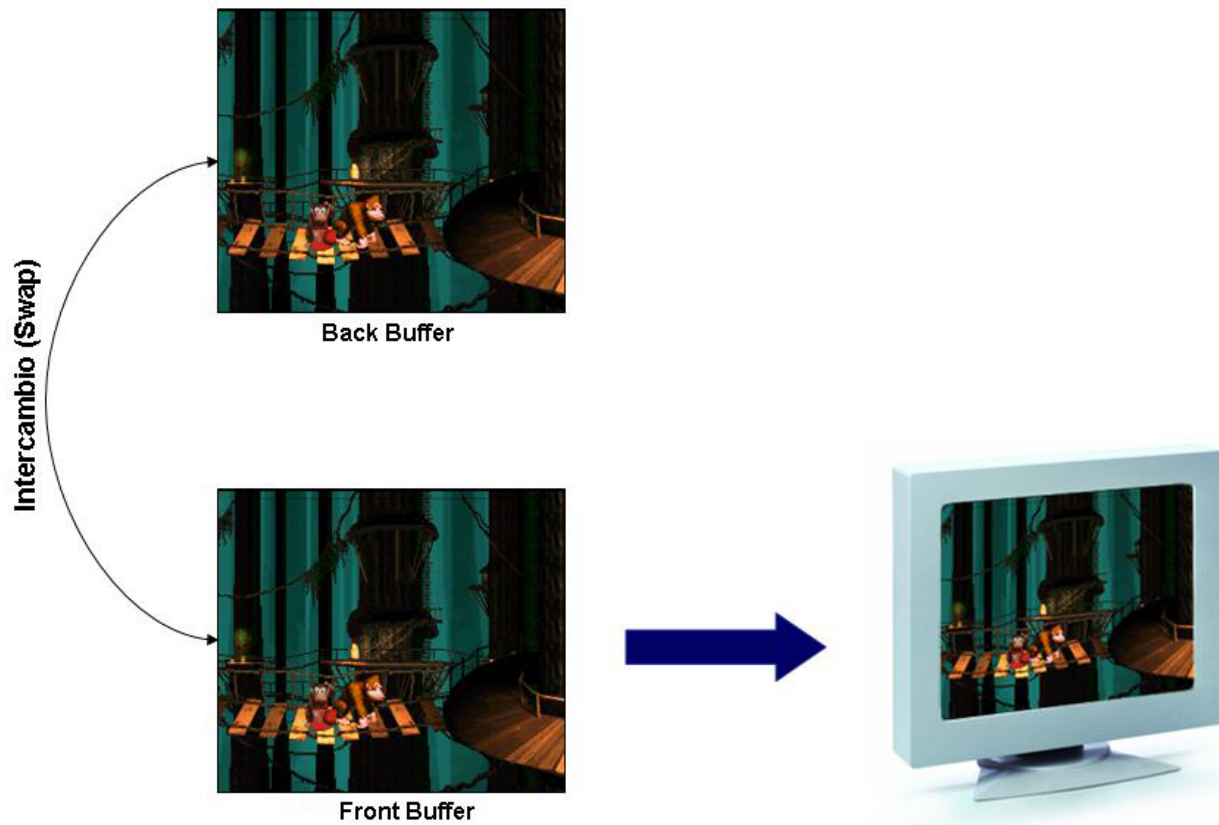
Para entender esta técnica, antes que todo debemos saber que el monitor ya sea CRT (Tubo de rayos catódicos) o LCD (Pantalla de cristal líquido), no dibuja la imagen en la pantalla instantáneamente, sino que lo hace pixel a pixel, partiendo desde la esquina superior izquierda hasta la esquina inferior derecha. Esto se realiza a una velocidad bastante rápida que el ojo humano no percibe, y a este tiempo, como vimos al principio, se le conoce como *frecuencia de refrescado*.

Cuando queremos mostrar una imagen en la pantalla, como ya sabemos colocamos los pixeles de la imagen en la memoria de video y durante el tiempo de refrescado el monitor leerá la información que hay en la Video RAM y la mostrará en la pantalla.

Ahora imaginemos que queremos mover un objeto por la pantalla, los pasos son simples, primero dibujamos la imagen en una posición, luego la borramos, y la dibujamos en la nueva posición, pero aquí aparecen algunos problemas, si a la mitad del refrescado cambiamos la imagen en la memoria de video, el monitor ahora obtendrá otra información de la imagen y probablemente la parte superior e inferior de la imagen no correspondan, es decir, veremos imágenes superpuestas y además un molesto

parpadeo. Una posible solución a esto, es esperar a que el monitor termine de refrescar la pantalla, para luego escribir en la Video RAM la nueva imagen. La mayoría de las APIs gráficas disponen de alguna función que espera el refrescado del monitor, evitando así el parpadeo y la superposición de imágenes, pero no del todo.

Para evitar totalmente el parpadeo, usamos esta técnica llamada **Double Buffering**, que consiste en tener *dos áreas de memoria* en la RAM (o Video RAM). Una de estas zonas se conoce como **front-buffer**, y corresponde a la que se muestra actualmente en pantalla y también tenemos el **back-buffer**, que es donde dibujamos los objetos que formarán la escena final. Estas áreas de memoria deben tener las mismas dimensiones del modo de video seleccionado.



Ahora tenemos dos opciones para esta técnica. Si el back-buffer se encuentra en la RAM, deberemos copiar todo el contenido al front-buffer, es decir, a la memoria de video para que veamos los cambios en el monitor. Para esto realizaremos un proceso de blitting, que dependiendo del modo de video usado, puede ser un poco costoso.

La otra posibilidad es que el back-buffer también se encuentre en la Video RAM, de ser así, no tendremos que realizar un blitting, sino que haremos algo más simple, intercambiaremos estas dos zonas de memoria, lo cual tiene un costo bastante bajo. A este proceso de intercambio del back-buffer con el front-buffer, se le conoce con el nombre de **Page Flipping**, y es lo que se suele usar hoy en día, ya que disponemos de

más RAM en las tarjetas de video. Además la función de Page Flipping, está implementada en todas las APIs gráficas, así que solo deberemos llamarla en el momento adecuado.

### **Dirty Rectangles**

Esta técnica es una forma de optimizar la anterior. Con Double Buffering constantemente estamos volcando el contenido completo de una zona de memoria a otra, pero puede haber ocasiones donde no este sucediendo ningún cambio en pantalla, o tal vez solo haya cambiado una pequeña parte de esta. Es aquí donde aparece la técnica, **Dirty Rectangles** (rectángulos sucios) y su funcionamiento es bastante simple. Copiaremos a la Video RAM solo las áreas de la pantalla que han cambiado, por lo tanto, cada vez que algún sprite cambie su posición, copiaremos a la memoria de video el área de un rectángulo que rodee al sprite (incluyendo el área donde se encontraba antes) y la colocaremos justo en las mismas coordenadas en la Video RAM. Pero no siempre es recomendable utilizar esta técnica, ya que podemos tener demasiados sprites moviéndose por la pantalla, y ya no sería óptimo estar copiando cada rectángulo a la Video RAM, ya que sería lo mismo que copiar la pantalla completa, en un caso así conviene solo usar la técnica Double Buffering.

### **Clipping**

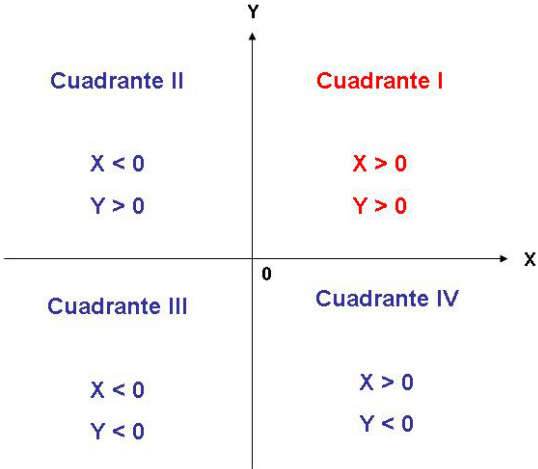
El clipping es un técnica bien sencilla, consiste en definir una área de recorte para la pantalla, es decir, todo lo que se dibuje fuera de esta área será ignorado y no se mostrará en pantalla. Normalmente esta área de recorte coincide con la resolución elegida para el videojuego, pero puede ser cualquier otra.

### **Sistema de coordenadas 2D**

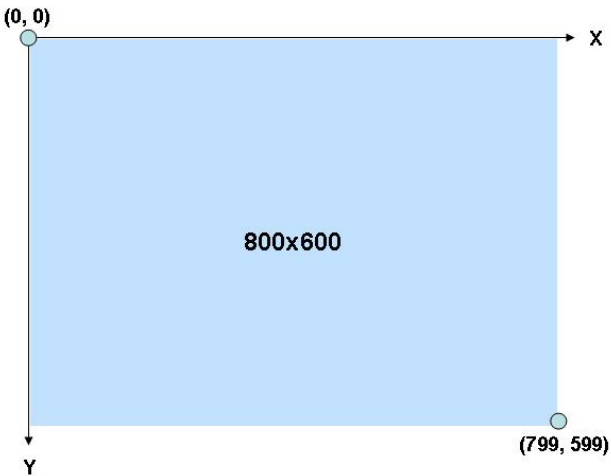
Es importante saber que cuando dibujamos algo en pantalla, siempre tendremos que informar la posición en la cual dibujaremos el objeto. Por defecto la posición de un objeto se encuentra en la esquina superior izquierda, pero algunos prefieren usar como punto de anclaje el centro, y en general puede ser cualquier otro punto. Calcular el centro del punto es tan simple como sumar a la posición x, la mitad del ancho del objeto y a la posición y, la mitad de la altura del objeto.

Nosotros estamos acostumbrados a trabajar en el cuadrante I del plano cartesiano, donde las coordenadas del eje "x" crecen hacia la derecha y las coordenadas del eje "y" crecen hacia arriba, pero al trabajar con gráficos en el computador, esto es relativamente distinto, ya que el eje y se invierte, es decir, ahora el mundo está al revés, las coordenadas en el eje "y" crecerán hacia abajo.

En la siguiente imagen podemos ver el plano cartesiano en su forma normal versus el plano cartesiano utilizado en la programación gráfica.



*Plano Cartesiano*



*El mundo al revés*

## **Sincronización en los videojuegos**

Lo ideal en cualquier videojuego, es que todos los objetos se muevan a la misma velocidad, independiente de la velocidad del computador donde se ejecute. Si no nos preocupamos por esto, y ejecutamos nuestro videojuego en un computador antiguo, por ejemplo, en un Pentium de 100 Mhz, probablemente el videojuego se vea muy lento, en cambio si lo ejecutamos en un computador con un procesador de última generación, un Pentium IV a 2.4 Ghz, se verá tan rápido que será imposible jugar.

Para solucionar este problema, disponemos de dos métodos.

### **Sincronización por Framerate**

El primer método, consiste en sincronizar el **framerate**, también conocido como **FPS** o **Frames per Second** (Frames por segundo). Al hablar de FPS, nos referimos a la frecuencia con que se ejecuta el ciclo principal de un videojuego en un segundo. A mayor cantidad de FPS obtendremos una mayor fluidez en las animaciones.

En el cine se considera que una velocidad de 24 FPS es suficiente para que el ojo humano perciba una animación fluida, pero en los videojuegos esta cantidad es demasiado baja. Valores adecuados son sobre 60 o 100 FPS.

El método es bastante sencillo, y lo primero que debemos hacer cuando comienza el ciclo del videojuego, es obtener el tiempo transcurrido hasta ese momento, que normalmente se mide en milisegundos. Luego procesamos lo relacionado al videojuego, ya sea entrada, IA, lógica del juego, detección de colisiones, dibujo de gráficos, etc. y antes de terminar el ciclo, creamos otro loop en el cual vamos obteniendo el tiempo transcurrido hasta ese momento, calculamos la diferencia de tiempo y verificamos si es menor a los FPS que buscamos. De esta forma cada vez que ejecutemos el programa en una máquina diferente, el programa esperará el tiempo adecuado para obtener los FPS.

Utilizando este método, en computadores más rápidos se verá más fluido, pero si lo ejecutamos en una máquina con un procesador mucho más antiguo del que usamos para desarrollarlo, lo más probable es que se vea bastante lento.

Claro, no todo podía ser perfecto, pero es por eso que los videojuegos piden algunos requerimientos mínimos.

Ahora veamos el código para este método de sincronización:

```
int t1, t2;           // Almacena los tiempos inicial y final
int fps=100;         // FPS esperados

while (!salir)
{
    t1=GetTime(); // Obtenemos tiempo inicial

    // Hacemos algo...

    do
    {
        t2=GetTime(); // Obtenemos tiempo final
    } while ((t2-t1) <= 1000/fps);

    // Si llegamos a este punto es porque ya paso el tiempo de espera
    // para que se cumplan los FPS esperados
}
```

### Sincronización por Tiempo

El segundo método consiste en sincronizar en base al tiempo. En este método no importa el framerate que posea el videojuego, pero aun así los objetos se moverán a la misma velocidad en todas las máquinas.

Básicamente, lo que debemos hacer es calcular la posición de un objeto de una forma distinta, en el método anterior si queríamos mover un objeto 5 pixeles por frame haríamos lo siguiente:

**$x = x + 5;$**

Y para mantener siempre la misma velocidad en varios equipos, esperaríamos un tiempo determinado al final del ciclo del videojuego.

Lo que haremos ahora será obtener el tiempo que ha transcurrido hasta el momento de calcular la nueva posición del objeto, y multiplicar ese tiempo por la velocidad. Es decir:

**$x = x + vx*dt;$**

Ahora la variable  **$vx$** , que almacena la velocidad, no será entera (int) sino que un numero flotante (float), ya que ahora lo multiplicaremos por un **delta t** de tiempo.

Imaginemos ahora que  **$vx$**  tiene el valor **0.005**, y el ciclo demora **1 segundo** en ejecutarse, el nuevo incremento será de:

```
x = x + 0.005*1000  
x = x + 5
```

Justo 5 pixeles, como en el caso anterior.

Veamos otro caso, ahora el ciclo se demora **10 milisegundos**, lo cual es más real, entonces:

```
x = x + 0.005*10;  
x = x + 0.05;
```

Ahora el incremento es menor, claro, debe ser así porque en tan solo 10 milisegundos se mueve 0.05 pixeles, entonces cuando pase 1 segundo se habrá movido:

**$(1000/10)*0.05 = 100*0.05 = 5$  pixeles**

Justamente lo que esperábamos, ahora no importan los FPS que existan en el videojuego, basándonos en el tiempo, los objetos se moverán a la misma velocidad en cualquier equipo.

Veamos esto en un simple código:

```
int t_new, t_old; // Almacena los tiempos actual y anterior  
int dt; // Diferencia de tiempos  
  
t_new=GetTime(); // Tiempo actual  
  
while(!salir)  
{  
    t_old=t_new; // El tiempo anterior  
  
    // Hacemos algo...  
  
    t_new=GetTime(); // Obtenemos tiempo actual  
  
    dt = t_new - t_old; // Calculamos diferencia de tiempos  
  
    x = x + vx*dt; // Calculamos nueva posición  
  
    // Dibujamos objetos...  
}
```

Este método se usa bastante en videojuegos 3D, ya que el framerate varía mucho en cada ciclo, dependiendo de la cantidad de objetos que se deban renderizar.

Pero este método también tiene su desventaja, a pesar de que los objetos se mueven siempre a la misma velocidad, en un computador más lento, el desplazamiento no se verá fluidamente. Esto se aprecia en el ejemplo que ya vimos, en el caso extremo de demorarse 1 segundo cada ciclo, cada vez que se deba mover un objeto, este aparecerá 5 pixeles más a la derecha, produciéndose un salto muy notorio. En computadores aun más lentos, se comenzará a ver un parpadeo en el desplazamiento de los objetos.

Pero como ya dijimos nuestro videojuego siempre tendrá unos requerimientos mínimos y tendremos que seleccionar el método que más nos acomode. Pero claramente el primer método, sincronización por framerate, es el más simple y el más extendido.

## **Resumen**

Hemos aprendido o repasado varios conceptos, que nos ayudarán a comprender mejor la programación gráfica que se usa en los videojuegos 2D.

Todos estos conceptos se pueden aplicar a cualquier biblioteca gráfica, muchas de las cosas que vimos ya vienen implementadas, como por ejemplo setear un modo de video, obtener un color, realizar operaciones de blitting, efectos de alpha blending, usar transparencias por color keys, transformaciones como rotación, escalado, traslación, page flipping, etc., por lo tanto no será necesario que escribamos código para esto, pero conviene conocer su funcionamiento.