

PROGRAMACION GRAFICA



**SISTEMA DE REPRESENTACION GRFICA**

**SIGLA:**

**Elc 102 - “SA”**

**NOMBRE:**

**Jhasmany Jhunnior Fernandez Ortega**

**GESTIÓN:**

**2 2020**

U.A.G.R.M.

F.I.C.C.T.

**TEMARIO**

1. **MODULO**
   1. Introducción a la programación gráfica 2D.
   2. Hardware y representación de imágenes.
   3. Subsistema de Vídeo.
2. **MODULO**
   1. Diferencias entre 3D y 2D
   2. Conceptos básicos de programación 2D.
   3. Flipping.
   4. Resumiendo.
3. **MODULO**
   1. Sistemas de coordenadas.
   2. Las superficies son rectangulares siempre.
   3. Clippers.
   4. Color Keys.
   5. Canal Alpha.
4. **MODULO I:**
   1. **Introducción a la programación grafica 2D**

Un problema bastante común para los que se inician en la programación de videojuegos es el paso de programar en la consola a hacer aplicaciones gráficas, saben que tienen que usar bibliotecas gráficas como DirectX, OpenGL, SDL, Pygame, Allegro, etc. Pero no saben como empezar con todo esto y como funciona. En la red hay documentación, pero está bastante dispersa y no sigue una línea voy a intentar cogiendo un poco de aquí y de allá y algo de aportación propia hacer una guía básica sobre los fundamentos de como trabajar con aplicaciones gráficas para los que se inician.

Esto no será para ningún lenguaje concreto será teoría aplicable a cualquier lenguaje y biblioteca de gráficos 2D.

* 1. **Hardware y representación de imágenes**

Vamos a comenzar repasando algunos conceptos básicos sobre las imágenes y su representación en un ordenador.

Una imagen puede ser representada como un conjunto de puntos. Cada uno de esos puntos tienen un color asociado. El número de puntos que representa una imagen puede variar segúna la definición que queramos obtener. Con cuantos más puntos sea representada una imagen más calidad tendrá y más espacio necesitaremos para almacenarla. Este aumento de tamaño provocará un aumento de tiempo de computación para que pueda ser mostrada. Esto se traduce en un mayor consumo de espacio de disco y que las transferencias de datos que hace que la imagen se nos muestre en la pantalla de nuestro ordenador estén más cargadas al tener más datos que intercambiar. Por este motivo es fundamental trabajar con una resolución idónea, buscando el equilibrio entre espacio y calidad.

Estos puntos a los que hacemos referencia se conocen como **píxeles** y determinan la resolución de una imagen. Los monitores tienen una limitación en cuanto al número de píxeles que pueden mostrar. Esta limitación es un máximo ya que siempre podrá mostrar un número menor de píxeles. Cada uno de estos píxeles tendrá asociado un color para los que se destinará un determinado número de bits (**bits por píxel** o bpp) que determinaran la calidad, medida en profundida de color, de la imagen. Los valores más comunes para el bpp son:

• 8 bits (256 colores)

Debemos usar una paleta para establecer los distintos 256 posibles colores.

• 16 bits (65,536 colores o Highcolor)

Existen distintas combinaciones para el uso de estos 16 bits. Las más comunes son:

• 5 bits para rojo, 6 bits para verde y 5 para azul. Se utiliza esta distribución porque el ojo humano distingue más colores verdes que otros.

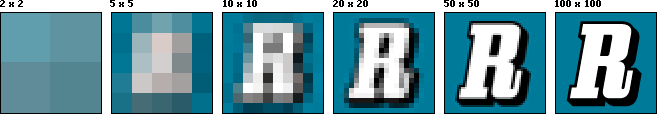
• 4 bits para rojo, 4 bits para verde, 4 para azul y 4 para transparencias. Este es un modelo más equitativo.

• 24 bits (16,777,216 colores o Truecolor)

Este es el modelo RGB puro. Se destinan 8 bits para cada color que cubren todo el espectro.

• 32 bits (16,777,216 colores)

Utiliza RGB más 8 bits destinados a canales alpha o transparencias.



Depende del sistema en el que trabajamos y para el que vaya destinado la aplicación definiran unas resoluciones de trabajo con una profundidad de color bien determinada. Como puedes suponer cuanto mayor sea el número de bits por píxel mayor será el tamaño de la imagen. Este es otro factor a considerar ya que necesitaremos la cantidad de bits indicada por el bpp para almacenar un sólo píxel. SDL nos proporciona funciones que nos permiten establecer estos modos de video así como consultar su compatibilidad con el sistema.

Un concepto importante es el de framebuffer que aparecerá varias veces en el tutorial. El framebuffer no es más que la zona de memoria que se corresponde con la imagen que mostrará el sistema en pantalla. Cada píxel tendrá una concordancia exacta en pantalla en una determinada posición. El formato del framebuffer suele ser un vector para simplificar la escritura sobre él. Al ser un vector unidimensional, a la hora de querer modficiar un píxel dado por unas coordenadas (x, y) tendremos que realizar una conversión de escala.

* 1. **Subsistema de Vídeo**

El subsistema de vídeo es la parte del sistema que nos permite interactuar con los dispositivos de vídeo. Generalmente el hardware de vídeo está compuesto por una tarjeta gráfica y un monitor o pantalla de visualización.

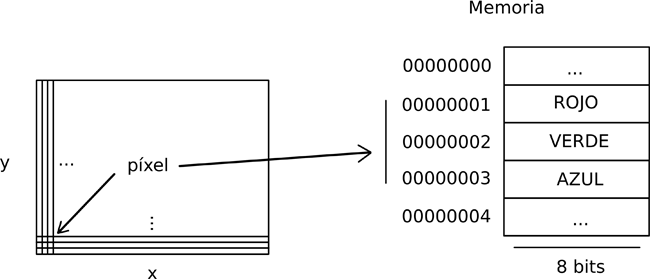
Existen numerosos tipos de tarjetas gráficas en el mercado pero todas ellas tienen características comunes que nos permiten trabajar con ellas independientemente del fabricante o del tipo de chip que monten dichas tarjetas. Poseen una unidad de procesamiento de gráficos y una memoria, ya sea compartida o dedicada, donde almacenar las estructuras o imágenes que se deben mostrar por pantalla.

El número de pantallas existentes en el mercado también es muy amplio pero no es algo que nos afecte a la hora de desarrollar un videojuego actualmente a no ser que lo hagamos para un dispostivo específico. En otros tiempos conocer la cantidad de colores que podía manejar un subsistema de vídeo era fundamental para diseñar cualquier aplicación para dicho sistema. Hoy en día la mayoría de los ordenadores que tenemos en casa son capaces de manejar el conocido como color real o true color a unas resoluciones que son prácticamente estándar.

Los principales aspectos que tenemos que tener encuenta de estos dispositivos son dos:

• La primera es la resolución de ambos dispositivos. Siempre deberemos trabajar a una resolución que sea admisible por el hardware en cuestión. Sería absurdo realizar un videojuego con un tamaño de ventana de 5000 x 5000 para un ordenador personal actual. Algo más avanzado el capítulo presentaremos las resoluciones más habituales.

El concepto de resolución de pantalla es bastante simple. Se trata del número de píxeles que puede ser mostrada por pantalla. Viene presentada, normalmente, por la ecuación x \* y donde x hace referencia al número de columnas de píxeles e y indica el número de filas de píxeles. Puedes observar las referencias en la imagen que se muestra a continuación.



• La segunda es la posibilidad de crear superficies en la memoria de la tarjeta gráfica. Algunas tarjetas gráficas tienen la memoria integrada en la placa base del ordenador por lo que comparten memoria con la CPU y emularán este comportamiento aunque almacenen el contenido en la memoria principal del sistema. El poder almacenar en dicha memoria las superficies que queramos mostrar es muy importante ya que concederá un mayor rendimiento a nuestra aplicación. Esta memoria es conocida como RAMDAC.

1. **MODULO II:**

**2.1. Diferencias entre 3D y 2D.**

Una biblioteca 3D se basa en polígonos. Estos polígonos están formados por vértices que tiene una serie de coordenadas para definirlos. Lo que hace nuestra biblioteca 3D es proyectar estos polígonos sobre un plano, en nuestro caso ese plano es la pantalla.

Cuando dibujamos esos polígonos en pantalla tenemos la opción de ponerles una imagen que se llama textura. A este proceso se le llama rasterización.

Se podría crear aplicaciones 2D utilizando una biblioteca 3D, bastaría con usar una proyección ortogonal.

Sin embargo, no todas las bibliotecas son así, existen otras que son exclusivas de 2D (SDL, Pygame…) en la que no existen los polígonos.

Estas bibliotecas se remontan a los ordenadores antiguos donde las gráficas no eran nada potentes y hacer calculos en como flotante que requieren los Bibliotecas 3D era inviable. Todo se hacía mediante procesador y les era más fácil hacer cálculos con operaciones enteras (esto ha cambiado ahora).

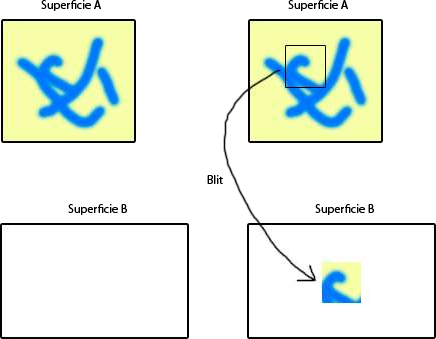
**2.2. Conceptos básicos de programación 2D**

Las bibliotecas 2D se basan en el concepto de superficies y de operaciones con superficies. Una superfice no es más que un espacio en la memoria donde guardar unos datos que se corresponden a una imagen. Las operaciones consisten en copiar trozos de una superficie origen a otra de destino.

Tomemos como ejemplo el Paint. Tenemos un lienzo en blanco en el que podemos dibujar, cargar una imagen desde el ordenador, etc.

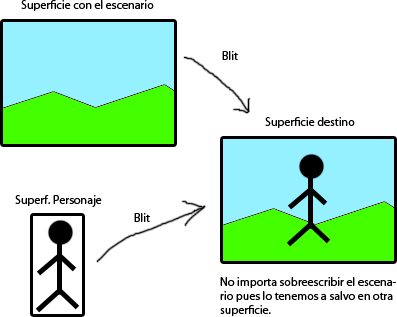
Ahora ejecutamos otro Paint, tendríamos otro lienzo en blanco independiente del primero. Estos dos lienzos serían dos superficies en las que podriamos dibujar o copiar parte de otra superficie.

Por ejemplo, cogemos un trozo de la imagen de uno de los Paints y la copiamos en el otro Dlienzo. Esta operación se llamo Blit y es la mas importante de una biblioteca 2D.



Al igual que en Paint, al hacer un Blit, lo que había en la superficie destino en la región que sobreescribimos se pierde para siempre. Al copiar sobre esa posición machacamos los datos anteriores. Esto es importante si tenemos, por ejemplo, un personaje moviéndose sobre un escenario. Al dibujar el personaje destruimos esa región del escenario, así que si el personaje se mueve, tendremos que volver a dibujar de nuevo el escenario por cada fotograma.

También de esto sacamos que el orden en que hagamos los blits importa, y mucho. Si algo tiene que quedar por encima, tiene que ser lo último que bliteemos.



Vale, con esto ya sabemos como montar imagenes entre varias superficies, pero ¿Cómo mostramos todo esto en la pantalla? Pues fácil, la pantalla no es más que otra superficie, una superficie especial que se muestra. Por tanto para mostrar algo en pantalla solo debemos hacer blits a esta superficie especial que se suele llamar screen.

La superficie de la pantalla (screen) suele tener un tamaño único durante todo el juego que es la resolución, como por ejemplo 640×480, este es el tamaño de nuestro lienzo y sobre él podemos hacer todos los blits que queramos.

**2.3. Flipping**

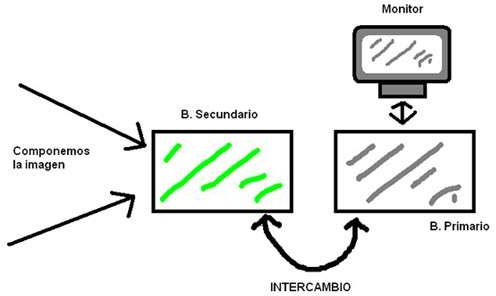
Vale, ahora tenemos una imagen que hemos compuesto sobre el Buffer Primario. Ya tenemos todo listo para que se muestre en el monitor. ¿Como hacemos para “actualizar” el monitor?

Depende, hay dos maneras de hacerlo, la fácil, y la difícil.

La fácil es llamar a la función de nuestra biblioteca que se encargue de dibujar y actualizar directamente todo a la superficie de nuestra screen. El problema de esto es que el monitor tiene un refresco de actualización, y si llamamos a esta función en el momento en que el monitor está dibujando algo, lo que veremos será un parpadeo bastante molesto.

Lo que queremos es que esa actualización se produzca justo en el momento en que el monitor ha terminado de dibujar un fotograma, y esta preparándose para el siguiente. A esto se le llama “sincronizarlo con el refresco vertical”, o en inglés, “VSYNC”.

Para hacer eso, lo que se usa es una técnica llamada “Double Buffer”. Consiste en que tendremos un buffer Secundario (o “BackBuffer”), además del primario. Nuestra imagen la compondremos en el secundario, y llegado el momento, intercambiaremos ambos buffers para que el secundario pase a ser el primario y viceversa, actualizando la imagen en el monitor. Este intercambio estará sincronizado con el refresco del monitor.



**2.4. Resumiendo:**

• Programación 3D != Programación 2D.

• Programación 3D == Punto Flotante, matrices, proyecciones + rasterizaciones.

• Programación 2D == Operaciones enteras, ints, copias de trozos de memoria de un lado a otro.

• Superficie: Espacio en memoria donde tenemos una imagen.

• Blit: Copy-paste de una región de una superficie sobre otra.

• Buffer Primario: Superficie que contiene lo que se dibuja en pantalla.

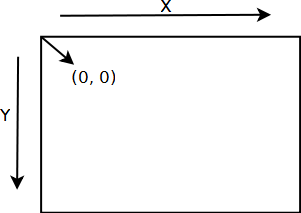
• Double Buffer: Algo raro que evita los parpadeos.

1. **MODULO III:**

**3.1. Sistemas de coordenadas**

Todas las superficies de las apis gráficas deben de tener un sistema de coordenadas, este se utiliza para saber en que parte se debe copiar una superficie a otra o que tamaño tiene una imagen que al fin y al cabo es lo que van a contener las superficies de nuestras aplicaciones.

El sistema de coordenadas tiene su origen en la esquina superior izquiera de la superficie, siendo ese pixel de la superficie la coordenada (0, 0) de la imagen.



ese pixel de la superficie la coordenada (0, 0) de la imagen.

Como vemos en la imagen tenemos dos ejes (x, y) siendo x el ancho de la superficie e y el alto. La imagen de arriba representa a cualquier superficie y como ya hemos dicho más de una vez una superficie es tanto una imagen como la misma pantalla.

**3.2. Las superficies son rectangulares siempre**

Es muy importante que se entienda que las superficies siempre son rectángulos. Y tu puedes decir, mentira, ¡El mismo Pacman es redondo! y yo te diré tu lo ves redondo, pero internamente no es más que un rectángulo que tiene un colo transparente (más adelante hablaremos del canal alpha, colorkey y demás).

Bien vamos a tratar con una superficie que represente una imagen, pongamos que tenemos esta imagen (No os riáis, no me llevo bien con el dibujo).



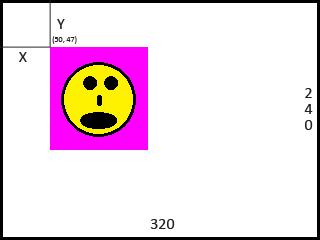
Está imagen podría ser la superficie de nuestro juego y tendría su coordenada (0, 0) en la parte superior derecha luego tendría un ancho y un elto en esta caso 98×103 por tanto el pixel inferior

derecho tendría la coordenada (98, 103). Otras coordenadas son las superior derecha sería la (98,

0) y la inferior izquierda sería la (0, 103). Creo que se capta la idea.

Ahora supongamos que queremos hacer un blit de nuestra superficie a la superficie screen, en otras palabras, copiar nuestra imagen en la pantalla. ¿Dónde se copiaría), pues en la coordenada que le pasemos a nuestra función que se encargue de hacer el blit.

Supongamos que tenemos una superficie screen de 320×240 y nuestra imagen y queremos copiarla en la coordenada (50, 47). Pues se copiará de la siguiente manera:

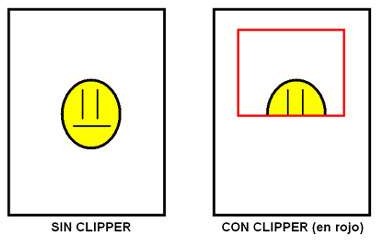


Como vemos la coordenada (0, 0) de la superficie origen se copia en la coordenada que elijamos de la superficie destino, es decir, la coordenada (0, 0) de nuestra superficie imagen la colocamos en la coordenada (50, 47) de la superficie de la pantalla. Por lo que para obtener la posición que ocupa el pixel inferior derecho de la imagen en la pantalla valdría una simple suma. (X+ANCHO\_IMAGEN, Y+ALTO\_IMAGEN), en nuestro caso sería (50+98, 47+103) que nos daría la coordenada del pixel final de la imagen.

Por lo general las bibliotecas gráficas existentes a parte del inicio se le puede indicar el final de lo que se quiere copar por lo que solo quedaría copiada esta área de la imagen. Vamos a hablar de los clippers.

**3.3. Clippers**

Lo que hace el clipper es definir el área de la superficie destino sobre el que podemos blittear. Todo lo que caiga fuera de ese área, no se dibuja.



El clipper de una superficie solo afecta a los blits que tengan como DESTINO a esa superficie. No afecta para nada al origen.

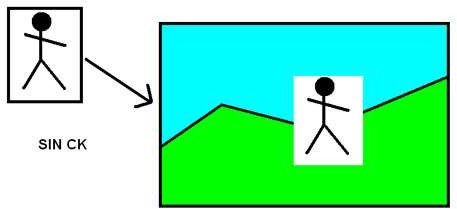
En la imagen de la izquierda dice “sin clipper”, pero eso no es completamente correcto. En realidad, siempre hay un clipper, sólo que por defecto su área abarca toda la superficie, dejando que bliteemos en todas partes.

La razón de que existan los clippers es sencillamente el evitar que al hacer un blit podamos escribir en zonas de memoria que no corresponden a la superficie. El rectángulo nunca puede extenderse más allá del limite de la superficie, así que ese problema desaparece automáticamente.

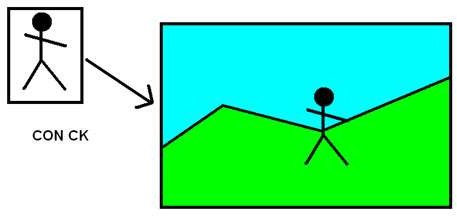
Pero además de eso, podemos darle más utilidades para optimizar el dibujado. Si sólo necesitamos actualizar una zona de la superficie destino, pero la región que estamos blitteando es enorme, usando un clipper nos evitamos escribir pixels que no necesitamos (o que no debemos, porque no queremos machacarlos).

**3.4. Color Keys**

Volvamos al problema que teníamos más arriba, donde decíamos que todos las superficies han de ser rectangulares, como el Pacman de más arriba. En realidad lo ideal sería que solo se dibujara nuestro sprite y no el área rosa de alrededor, para ello se usa el colorkey, esto consisten en definir un color de la imagen como transparente. Todo el color de una imagen marcado como colorkey se volverá transparente. Pongamos un ejemplo:



Como vemos el recuadro blanco queda muy mal, la solución es definir que en nuestra superficie de origen el colorkey se la imagen sería el blanco, por lo que nos quedaría:



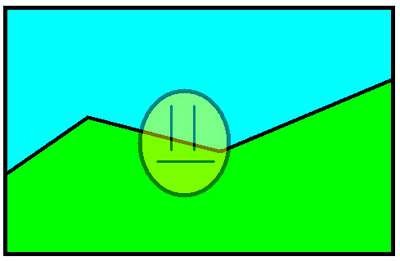
Como vemos mucho mejor, pero no olvidemos dos cosas importantes.

1. Aunque el rectángulo blanco sea transparente sigue estando ahí y la esquina superior izquieda de nuestra imagen sigue siendo la que era antes, aunque no se vea.

2. El colorkey hace que TODOS los píxeles de ese color se vuelvan transparentes por lo que debes asegurarte de que el color que uses como colorkey no forma parte de la imagen. Se suelen usar colores poco comunes como el (255, 0, 255) que es el rosa chillón de arriba, pero si tu imagen tiene este color basta con usar otro.

**3.5. Canal Alpha.**

Por último mencionar el canal Alpha, consiste en definir un grado de transparencia para la imagen.



porque en apis 2D que no usen tratamiento por gráficos 3D como OpenGL o DirectX usar el canal

alpha gasta muchos recursos pues no solo hay que leer la imagen de origen sino también el destino para la mezcla de colores haciendo que en ocasiones pueda ser un verdadero cuello de botella para el rendimiento.