UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Implementación básica de primitivas gráficas de dibujo sobre el "raster" de la NDS

Apellidos, nombre	**Agustí Melchor, Manuel** (magusti@disca.upv.es)
Departamento	**Dpto. de Ing. De Sistemas y Computadores**
Centro	**Escola Tècnica Superior d'Enginyeria Informàtica** Universitat Politècnica de València

1 Resumen de las ideas clave

En este artículo vamos a ver cómo se puede dibujar con primitivas gráficas básicas como líneas y círculos en las pantallas de la *Nintendo DS* (NDS) usando el concepto de *raster* y la técnica del acceso directo a la memoria de vídeo.

El ejemplo básico que nos inspira apareció en el sitio web de *Drunken Coders*[1] [1], un grupo de desarrolladores de la escena *homebrew* para el desarrollo de aplicaciones sobre NDS: esto es, que no utilizan el SDK oficial propuesto por el fabricante de la consola, que tiene unas restricciones tremendas sobre su publicidad y uso. Esperamos que este trabajo sirva para reconocer el tremendo esfuerzo de estos grandes desarrolladores, máxime en las épocas en que no había nada documentado. Su trabajo ha servido de inspiración y de fuente a muchos. ¡Gracias!

Ya no es posible consultar el tutorial original en la red, así que el presente artículo ofrece una versión de ese trabajo, actualizada a las versiones existentes de las librerías sobre las que se basa y ampliada, puesto que el segundo ejemplo está solo esbozado en el tutorial original. De todas formas, dejaremos los comentarios originales para que quede constancia del trabajo original. Para facilitar el seguimiento de este trabajo se han dispuesto todos los elementos del proyecto que aquí se comenta en *GitHub* [2]. Nos ocuparemos en este artículo de comentar las acciones necesarias para implementar las operaciones de dibujo básicas de línea y círculo en las pantallas de la NDS a nivel de píxel.

2 Objetivos

Una vez que el lector haya leído con detenimiento este documento:

- Será capaz de identificar el concepto de imagen en mapa de bits (*raster* o *bitmap*) con un modo de trabajo de la NDS.
- Será capaz de identificar las instrucciones que permiten el acceso directo a la memoria de vídeo en la plataforma NDS.
- Dispondrá de unos ejemplos de código que se proponen como ejercicios de iniciación al acceso a los píxeles de la pantalla.

No es un objetivo instalar las herramientas que permiten la creación del ejecutable así como la carga del mismo en la consola, de hecho, nosotros utilizaremos un emulador (*DeSmuMe* [4]) para generar las capturas. Para ello se puede recurrir a los trabajos de [5] o [6].

---

[1] Publicado originalmente en la URL <http://drunkencoders.com/ >, ahora alberga un enlace al contenido anterior al tutorial en que nos hemos inspirado. En *Github* <https://github.com/drunken-coders > ha aparecido un repositorio con la leyenda “Breaking new ground in console homebrew development”. Esperamos que puedan recuperar allí todos los ejemplos que habían conseguido elaborar en estos años atrás.

3 Introducción

La NDS puede representar las imágenes en sus pantallas en formato ***bitmap* o *raster*** (de mapa de bits*)*, esto es, que internamente se representa como una matriz de puntos o píxeles, Figura 1a[2]. Cada uno de estos puntos es capaz de almacenar la información de color en base a un espacio de representación de color. Este es generalmente el RGB, que utiliza tres valores: uno para el rojo, otro para el verde y otro para el azul. Son los colores primarios de este espacio de color aditivo. Esto quiere decir que estas imágenes tienen una resolución (espacial y de color) establecida en el momento de su creación, como se hace en una aplicación típica de dibujo en computador, en una impresora o al hacer una fotografía con una cámara digital.

Esto supone por un lado una optimización y, también, una restricción de partida, porque impone un límite en la posibilidad de representar una imagen sobre un área mayor de la que se pensó en principio y, por tanto, una situación en la que se observe el "pixelado" en la imagen (véase la Figura 1b): al utilizar una imagen (en mapa de bits) en una resolución mayor que la que se usó para crearla aparece el efecto de escalones o trazos no continuos debidos a la discretización del espacio que impone el *raster*, a efectos prácticos es un límite para el uso de esa imagen.

a) *b)*

Figura 1: Raster: (a) visualización gráfica del concepto y (b) ejemplo de pixelado.

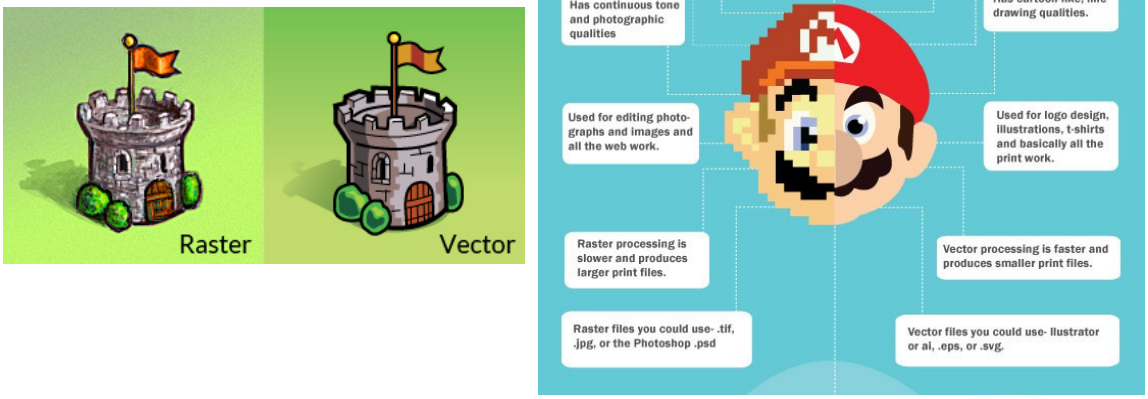
No todas las aplicaciones de dibujo permiten el trazo libre, por lo que hay una parte de estas aplicaciones que permiten dibujar utilizando solo un conjunto de operaciones, sus propias primitivas de dibujo, como líneas, curvas, polígonos, degradados, …. La Figura 2a muestra una imagen[3] en la que se pueden ver las típicas diferencias al observar el mismo motivo creado a base de "pintar" los puntos de la imagen (en formato de mapa de bits o *raster*) frente a pintar usando operaciones de dibujo vectorial.

En resumen, el *raster* no es el único formato para especificar una imagen, si se quieren ampliar estos conceptos se recomienda ver las fuentes que se citan en este artículo. P. ej. podemos ver una comparativa en la Figura 2b. Hay que pensar dónde va a ser usada una

[2] Se puede ampliar este concepto en la definición de rasterzación que muestra ese ejemplo, vease <https://en.wikipedia.org/wiki/Rasterisation>.

[3] Imagen obtenida de <http://www.xpertprints.com/portfolio/>.

imagen para decidir acerca del número de colores de esta, su resolución o los posibles formatos de ficheros en que se almacenará.



a) *b)*

*Figura 2: Comparativa entre formato "raster" y vectorial: (a) ejemplo de un mismo motivo creado en modo "*raster*" o en modo vectorial y (b) resumen de características de ambos formatos. Fuente [3].*

Las videoconsolas, como la NDS, especifican la resolución de sus pantallas y, dada la limitada disponibilidad de recursos de estas plataformas portables, es interesante ver cómo acceder y usar sus posibilidades gráficas. Vamos a ver que el *hardware* de la NDS utiliza una parte de su memoria central (RAM) como "memoria de vídeo" (VRAM y también denominada *framebuffer);* esta área de la RAM mantiene el mapa de bits que se muestra en las pantallas. El *hardware* de vídeo genera, a partir de esa información, la señal que se visualiza en las pantallas.

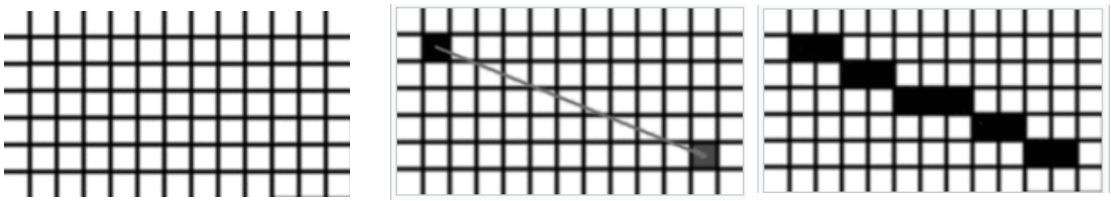
La NDS permite dibujar en pantalla en formato *raster*, lo que significa que es posible dibujar en sus pantallas píxel a píxel. Vamos a ver ejemplos de código que implementa el uso de líneas y círculos.

4 Acceso al *raster* en la NDS

Al trazar una línea recta sobre papel con un lápiz, no existen impedimentos para colocar una regla y seguir una línea completamente recta. Dibujar líneas y círculos puede parecer extremadamente simple, pero veamos que conectar dos puntos en una pantalla 2D por una serie de píxeles (*raster*) ha sido objeto de mucha investigación. El lector interesado puede acudir a las referencias que le llevarán a profundizar en los conceptos formales que necesite explorar con más detalle.

El problema radica básicamente en la discretización del espacio que supone el *raster*. Echemos un vistazo de cerca a cómo se ve una línea en la pantalla de una computadora para tener una idea de lo que decimos. Como ya hemos visto en la Figura 1, la línea solo puede moverse en pasos discretos de píxeles, para representar una línea podemos avanzar

entre casillas vecinas en cualquiera de las dos dimensiones. Mientras mantengamos la conexión, se formará una línea. La solución viene de la mano del algoritmo de Bresenham[4], quien propuso la forma de pintar en una impresora utilizando la idea de *raster* y optimizando los cálculos con operaciones de tipo entero que están implementadas en todos los equipos *hardware*. Su trabajo define que cuando se dibuja una línea sobre un *raster (*Figura 3a), habremos de iterar a través de, p. ej., las columnas (X) y cambiar de fila (Y) en función de la pendiente. Así, una línea (Figura 3b) que va del punto (*x1,y1*) en la parte superior izquierda del *raster*, al punto (*x2,y2*) en la parte derecha inferior; por lo que cambia más en la dirección *X* que en la *Y*: ha de avanzar diez casillas en horizontal, mientras que en vertical solo cuatro. Es decir, el trazado de una línea sobre este tipo de soportes, donde forzosamente debe ocuparse como unidad mínima un cuadro del *raster* (como equivalente de un punto), siempre será una aproximación de la línea ideal y, por tanto, siempre contiene errores *(*Figura 3c), que son la consecuencia del paso de valores continuos a valores discretos, el paso de lo analógico a lo digital.



a) *b)* *c)*

Figura 3: Idea gráfica del algoritmo de Bresenham: (a) rejilla o raster, (b) recta continua entre dos puntos y (c) recta discreta entre esos dos puntos. Fuente <https://es.wikipedia.org/wiki/Algoritmo_de_Bresenham>.

$$m = \frac{ydiff}{xdiff};\ donde\ xdiff = x_1 - x_2;\ ydiff = y_1 - y_2 \qquad \textit{Ec. 4.1}$$

Para calcular la pendiente, normalmente haríamos algo como la Ec. 4.1, donde hemos nombrado como *ydiff* y *xdiff* a las diferencias en las coordenadas de los ejes vertical y horizontal, respectivamente. Si tuviéramos resolución infinita, como sucede al dibujar sobre papel, primero trazaríamos un píxel en (*x1*, *y1*) y luego moveríamos una *X* hacia la derecha. Si esto fuera una pantalla infinita, también moveríamos *m* píxeles hacia abajo y trazaríamos el píxel. Debido a que nuestra pantalla real es finita, lo mejor que podemos hacer es (Figura 3c) mover un píxel en X y cero en Y. Si continuásemos, moveríamos otro píxel en X y otro tanto de *m* hacia abajo en Y, pero puesto que tenemos una pantalla discreta y no podemos avanzar un valor fraccionario de un píxel, no haríamos ningún cambio en Y. Finalmente, mientras nos movemos por tercera vez en X llegamos a un punto en el que la pendiente es mayor de uno y eso nos indica que debería moverse un píxel completo hacia abajo en la dirección de Y.

Este proceso es la base del algoritmo de *Bressenham*. Hay que tener en cuenta que los puntos pueden estar en otras disposiciones y aparecen

[4] Véase en <https://es.wikipedia.org/wiki/Algoritmo_de_Bresenham>.

cuestiones de signos y otros detalles que obviamos por brevedad en la exposición.

4.1 Dibujo de líneas

Para nuestra plataforma, estas ideas se pueden implementar como muestra el código del Listado 1 y Listado 2, que es nuestra propia implementación del código de [1]. La salida de este ejemplo se puede ver en la Figura 4. Observaremos que solo los trazos completamente verticales u horizontales aparecen perfectamente representados.

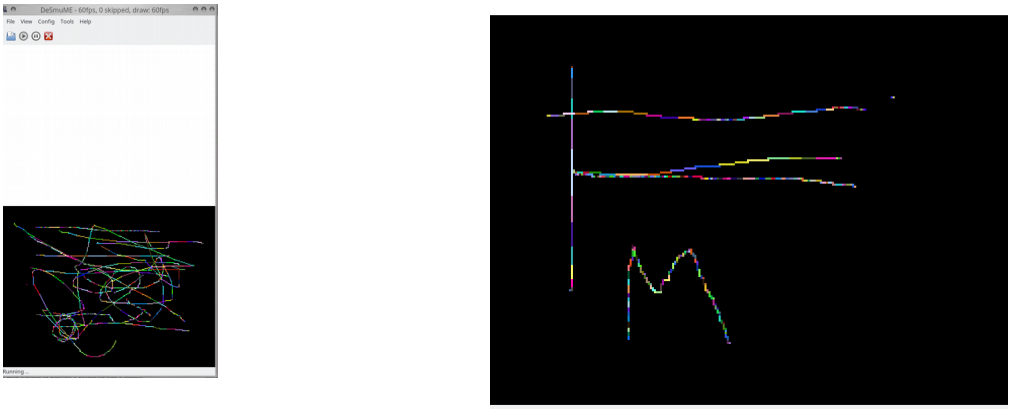


Figura 4: Ejemplos de salida del código de "lines" (Listado 1 y Listado 2).

```
1.  #include <nds.h>
2.  #include <stdio.h>
3.  void  DrawLine(int x1, int y1, int x2, int y2, unsigned short color);
4.  int main( void ) {
5.    touchPosition touch;
6.    int oldX = 0, oldY = 0;
7.    videoSetMode(MODE_FB0);  vramSetBankA(VRAM_A_LCD);
8.    lcdMainOnBottom();
9.    while(1)     {
10.     scanKeys();    touchRead(&touch);
11.     if(!(keysDown() & KEY_TOUCH) && (keysHeld() & KEY_TOUCH)) {
12.       DrawLine(oldX, oldY, touch.px, touch.py, rand());
13.     }
14.     oldX = touch.px;  oldY = touch.py;
15.     swiWaitForVBlank();
16.   }
17.   return 0;
18. } // Main
19. ...
```

Listado 1: Listado de "lines", parte 1. Código extraído de [1].

El programa principal establece el modo de vídeo de una de las pantallas en modo *framebuffer* (línea 7) para poder dibujar sobre ella utilizando la idea del *raster* y asignándole el espacio en la VRAM y la pantalla inferior de la NDS. El programa principal estará continuamente (líneas 9 a la 16) comprobando si se pulsa sobre la pantalla y se mantiene la pulsación, para trazar una línea desde ese punto hasta que se deje de tocar la pantalla. La última posición será la que se utilizará como origen para la siguiente línea. Finalmente, actualizamos las *oldx* y *oldy* y repetimos.

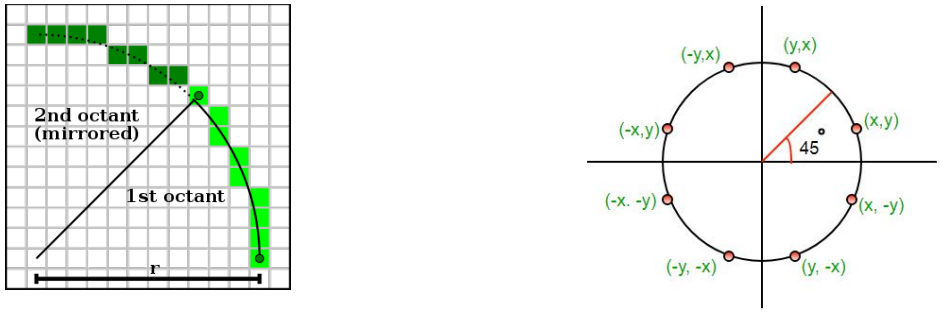
```
20.    ...
21.    void  DrawLine(int x1, int y1, int x2, int y2, unsigned short color) {
22.      int yStep = SCREEN_WIDTH, xStep = 1;
23.      int xDiff = x2 – x1, yDiff = y2 - y1;
24.      int errorTerm = 0, offset = y1 * SCREEN_WIDTH + x1,
25.      if (yDiff < 0)    {   //need to adjust if y1 > y2
26.         yDiff = -yDiff;          //absolute value
27.         yStep = -yStep;       //step up instead of down
28.       }
29.      if (xDiff < 0)    {   //same for x
30.         xDiff = -xDiff;       xStep = -xStep;
31.       }
32.      if (xDiff > yDiff)   {  //case for changes more in X than in Y
33.         for (i = 0; i < xDiff + 1; i++)    {
34.               VRAM_A[offset] = color;
35.               offset += xStep;       errorTerm += yDiff;
36.               if (errorTerm > xDiff)   {
37.                 errorTerm -= xDiff; offset += yStep; }
38.             }
39.       }//end if xdiff > ydiff
40.      else    {   //case for changes more in Y than in X
41.         for (i = 0; i < yDiff + 1; i++)    {
42.               VRAM_A[offset] = color;
43.               offset += yStep;       errorTerm += xDiff;
44.               if (errorTerm > yDiff) { errorTerm -= yDiff; offset += xStep;}
45.             }
46.       }
47.    } // DrawLine
```

Listado 2: Listado lines, parte 2. Código extraído de [1].

4.2 Dibujar círculos

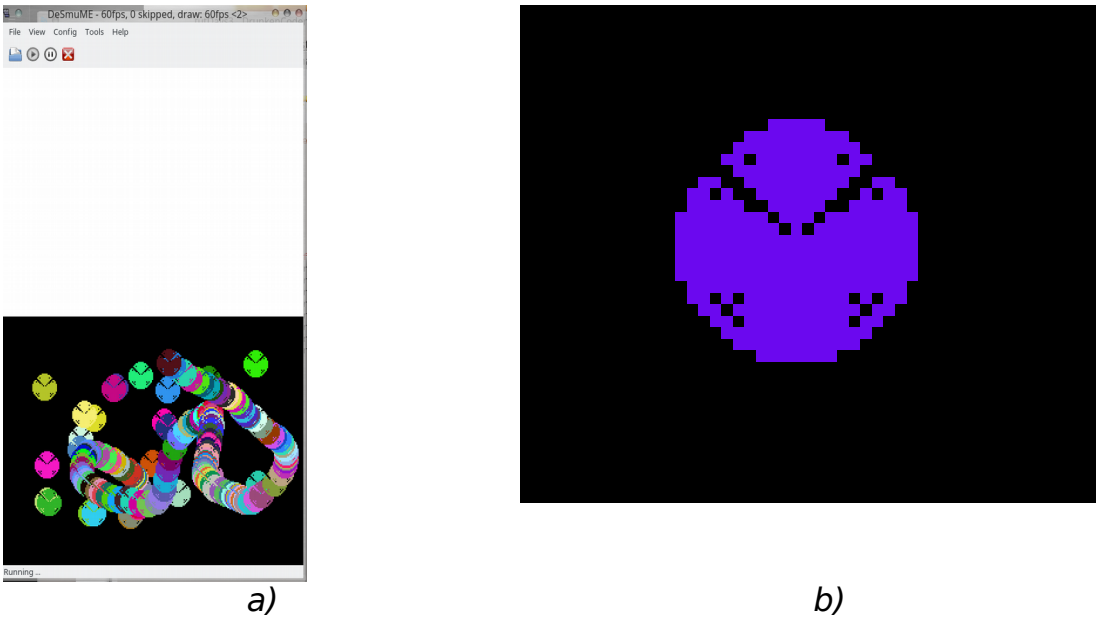
El tutorial original ofrece dos trozos de código en lenguaje Pascal para realizar esta operación. Aquí los hemos convertido a C y hemos añadido algunas ampliaciones, aprovecharemos la ocasión para modificarlos y, así, explorando sus opciones, jugar un poco con ellos.

Para dibujar un círculo, *Bresenham* se basó en el "Algoritmo del punto medio para circunferencias"[5]. Calculando los puntos del primer octante del círculo (Figura 5a) se pueden pintar los demás por simetría (Figura 5b).



a) *b)*

Figura 5: Para dibujar un círculo: (a) discretización de los puntos en el "raster" y (b) la situación se repite en cada octante.



a) *b)*

Figura 6: Ejemplo de salida de circles: (a) captura de las dos pantallas y (b) ampliación de uno de las círcunferencias.

[5] "Bresenham" <https://www.geeksforgeeks.org/bresenhams-circle-drawing-algorithm/> y "Mid-Point Circle Drawing Algorithm" <https://en.wikipedia.org/wiki/Midpoint_circle_algorithm>.

En la NDS y a partir del pseudocódigo de [1], hemos nuestra propia implementación del código de dibujo de círculos en los Listado 3 y Listado 4. La salida de este ejemplo se puede ver en la Figura 6a donde hemos ido pintando varias círcunferrncias y también podemos ver una ampliada (Figura 6b). Como habrá podido observar el lector, en esa ampliación se ven "agujeros" o imperfecciones de la implementación del algoritmo. Lo dejaremos a la curiosidad del lector investigar el motivo: los puntos del círculo se pintan con las líneas 15 a la 22 del Listado 3 (como decía la Figura 5b) y para rellenar el interior se van pintando cuatro líneas con los *DrawLine* de las líneas 24 a la 27. ¿Se imagina, el lector, cómo añadir un parámetro más para decidir si dibujar círculos o circunferencias a voluntad?

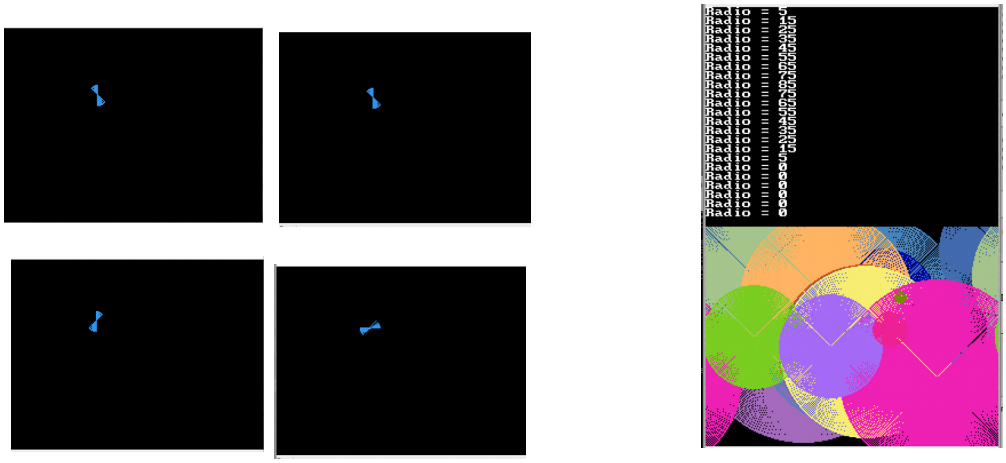
```
1.  #include <nds.h>
2.  #include <stdio.h>
3.  void  DrawLine(int x1, int y1, int x2, int y2, unsigned short color) {
4.    // Este código es el mismo de Listado 2, por lo que se omite
5.  } // DrawLine
6.  int esferic = 8;
7.  void DrawPixel(int X, int Y, unsigned short Color) {
8.    VRAM_A [X + Y * SCREEN_WIDTH] = Color
9.  } // DrawPixel
10. void DrawCircle (int rayon, int x_centre, int y_centre, unsigned short Color) {
11.  int x, y, m;
12.  x = 0; y = rayon;
13.  m = 5 – 4*rayon;.      // Place on the top of the circle  initialisation
14.  while (x <= y) {                    // while we are in the second half
15.   DrawPixel(  x+x_centre,  y+y_centre, Color );
16.   DrawPixel(  y+x_centre,  x+y_centre, Color );
17.   DrawPixel( -x+x_centre,  y+y_centre, Color );
18.   DrawPixel( -y+x_centre,  x+y_centre, Color );
19.   DrawPixel(  x+x_centre, -y+y_centre, Color );
20.   DrawPixel(  y+x_centre, -x+y_centre, Color );
21.   DrawPixel( -x+x_centre, -y+y_centre, Color );
22.   DrawPixel( -y+x_centre, -x+y_centre, Color );
23.
24.  DrawLine( x+x_centre, y+y_centre, -x+x_centre, -y+y_centre, Color );
25.  DrawLine( y+x_centre, x+y_centre, -y+x_centre, -x+y_centre, Color );
26.  DrawLine(-x+x_centre, y+y_centre,  x+x_centre, -y+y_centre, Color );
27.  DrawLine(-y+x_centre, x+y_centre,  y+x_centre, -x+y_centre, Color );
28.  if (m > 0) {  y = y – 1;  m = m - esferic*y; // m - 8*y;  }
...
```

Listado 3: Listado del código "circles", parte 1.

```
    ...
29.   x = x + 1;
30.   m = m + 8*x + 4;
31.  } // while
32. } // DrawCircle
33. #define MIN_RADIO 5
34. #define MAX_RADIO 100
35. void borrarPantalla() {
36.   int i;
37.   for (i = 0; i < SCREEN_WIDTH*SCREEN_HEIGHT; i++) VRAM_A[i] = 0;
38. } // borrarPantalla
39. int main( void ) {
40.   touchPosition touch; int radio = 5;
41.   consoleDemoInit(); 61.              // inicializa una pantalla en modo texto
42.   videoSetMode(MODE_FB0);63.   // y la otra en modo “raster “( “framebuffer “)
43.   vramSetBankA(VRAM_A_LCD);
44.   lcdMainOnBottom();
45. while(1)     {
46.     scanKeys();  touchRead(&touch);
47.     if(keysDown() & KEY_UP) {
48.       radio = (radio+10) % MAX_RADIO;   printf("Radio = %d\n", radio);    }
49.     if(keysDown() & KEY_DOWN) {
50.       radio = (radio <= MIN_RADIO? 0 : (radio-10)); printf("Radio = %d\n", radio);}
51.     if(keysDown() & KEY_A) { borrarPantalla();  printf("\n");  }
52.     if(keysDown() & KEY_X) {
53.       esferic = (esferic <= 0? 0 : (esferic-10)); printf("Esferic = %d\n", esferic);  }
54.     if(keysDown() & KEY_Y) {
55.       esferic = (esferic >= 100? 100 : (esferic+10));
56.       printf("Esferic = %d\n", esferic);    }
57.     if(!(keysDown() & KEY_TOUCH) && (keysHeld() & KEY_TOUCH)) {
58.       DrawCircle( radio, touch.px, touch.py, rand() );   }
59.     swiWaitForVBlank();
60.   }
61.   return 0;
62. } // Main
```

Listado 4: Listado del código “circles”, parte 2.

La Figura 7 nos muestra qué hacen las instrucciones de las líneas 24 a la 27 del Listado 3 o como se varía el radio del círculo en las líneas 47 a la 50 del Listado 4.



a) *b)*

Figura 7: Detalles del ejemplo “circles”: (a) Salida de cada uno de los cuatro “DrawLine” del Listado 3 y (b) modificando el radio.

5 Conclusiones y cierre

En este artículo hemos visto que la NDS puede trabajar en modo *raster*, lo que nos permite acceder a cada punto de la pantalla y asignarle un color. A partir de esa idea y con las particularidades que impone tener un tamaño discreto y fijo de puntos (los píxeles) hemos visto dos ejemplos de código que nos permiten una iniciación al dibujo con las primitivas básicas de línea y círculo.

Esperamos que el lector se anime a ponerlos en marcha y explorar con sus propias ideas el código de los proyectos que hemos mostrado y que está disponible en [2].

6 Bibliografía

[1] Dovoto. (2015). Day 3 – Raster.. Ubicado originalmente en <http://www.drunkencoders.com >, no está disponible actualmente.

[2] M. Agustí, (2020). Repositorio del proyecto “NDS-homebrew-development”. Disponible en <https://github.com/magusti/NDS-hombrew-development>.

[3] N, Khattar. (2016). Super Mario themed Infographic: Raster vs Vector Image. Disponible en <.https://www.behance.net/gallery/42717355/Super-Mario-themed-Infographic-Raster-vs-Vector-Image>.

[4] DeSmuME. Página web del proyecto. Disponible en <http://desmume.org/>.

[5] J. Amero (Patater). (2008). Introduction to Nintendo DS Programming. Disponible en <https://patater.com/files/projects/manual/manual.html>.

[6] GBATEK. Gameboy Advance / Nintendo DS - Technical Info. Disponible en <https://www.akkit.org/info/gbatek.htm>.