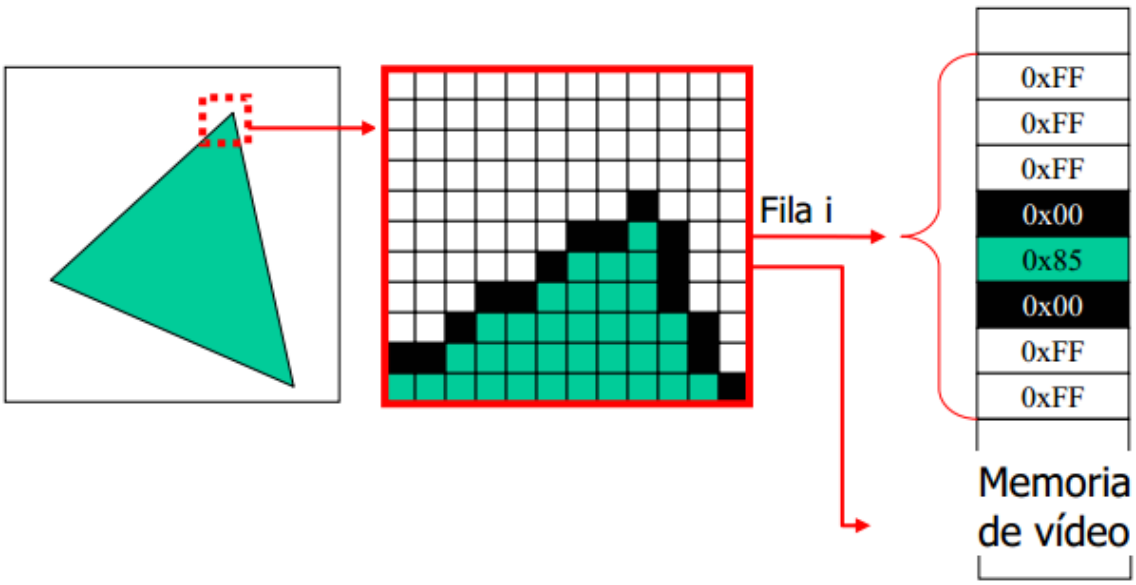
Dibujo de figuras elementales

J. Fco. Jafet Pérez L.

Primitivas 2D

Algunas formas geométricas son consideradas primitivas por su básica constitución en las partes que la conforman, se conocen también con el nombre de primitivas geométricas. En un plano 2D podemos trabajar con primitivas tales como puntos, líneas, círculos, rectángulos, elipses, polígonos, etc.

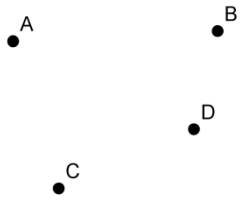
En los gráficos de mapa de bits, las imágenes vienen definidas por la intensidad de sus pixeles. Los objetos presentes en una imagen se componen de primitivas simples. El sistema gráfico dibuja estas primitivas transformándolos en pixeles (Rasterización)



Los métodos de conversión deben ser lo más eficientes posible.

Primitiva Punto.

La primitiva "**Punto**" es la más sencilla: Basta con especificar la intensidad de color deseada de la posición del pixel a pintar.



Implementación en javascript – html. (Ejemplo 1)

El siguiente ejemplo muestra un pixel en color rojo en la posición 120, 100 de un lienzo (canvas) html. El punto que se muestra no mide 1x1 pixel por cuestiones de claridad le he puesto las dimensiones de 3x3 pixeles (En monitores con resolución muy alta, 1 pixel es muy pequeño y podría perderse a los ojos del usuario).

He dividido la implementación en dos archivos pixel.html y pixel.js con la finalidad de separar la presentación de la lógica de la solución, es decir, en el documento html especificamos únicamente la interfaz formada por la etiqueta <canvas> y en el archivo .js incluimos el código para pintar el pixel.

---------- **pixel.html**

```
<!DOCTYPE html>
<html>

  <head>
    <title>Pixel en la posición 120, 100</title>
  </head>

  <!-- Al cargar la página se llamara a la rutina para pintar -->
  <body onload="dibujar()">

    <!-- Archivo pixel.js con la rutina para pintar el pixel utilizando javascript -->
    <script src="pixel.js"></script>

    <canvas id="lienzo" width="400" height="400"> <!-- Lienzo para pintar el punto -->
    </canvas>

  </body>

</html>
```

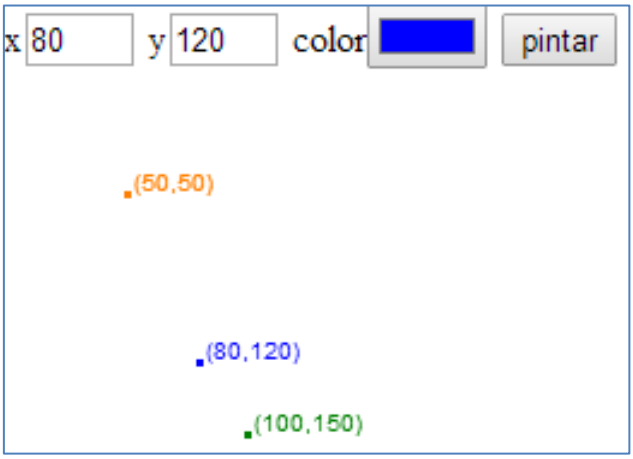
---------- **pixel.js**

```
function ponerPixel(contexto, x, y, color){

  contexto.fillStyle = color; //configura el color para pintado

  //pintar un punto (debe ser 1x1, para mejorar su visualización es de 3x3)
  contexto.fillRect( x, y, 3, 3 );

}


function dibujar(){ //Esta función es llamada al cargar el documento html

  var canvas = document.getElementById("lienzo"); //accedemos al lienzo de dibujo

  var contexto = canvas.getContext("2d"); //obtenemos el contexto 2d del lienzo

  ponerPixel(contexto, 100, 120, "#FF0000"); //pintamos el pixel 100, 120 en rojo

}
```

Especificar coordenadas y color del pixel. (Ejemplo 2)

El ejemplo anterior muestra un pixel en una posición y color establecidos de manera directa en el código (120, 100, rojo), sin embargo, podríamos solicitar al usuario dicha ubicación y color.

El siguiente ejemplo utiliza campos de entrada en un formulario html para solicitar la información del punto que se desea graficar. El resultado es una interfaz como esta:



De nuevo, he dividido la implementación en dos archivos pixelxy.html y pixelxy.js con la finalidad de separar la presentación de la lógica de la solución.

---------- **pixelxy.html**

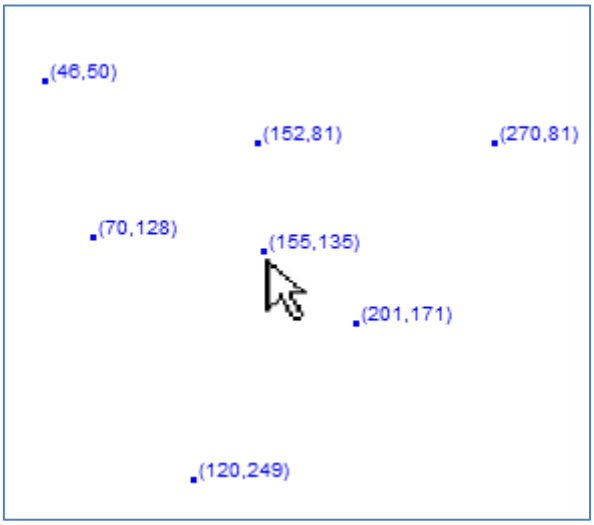
```
<!DOCTYPE html>
<html>

  <head>
    <title>Pixel en una posición x,y indicada por el usuario</title>
  </head>

  <body>

    <!-- Archivo pixelxy.js con la rutina para pintar el pixel utilizando javascript -->
    <script src="pixelxy.js"></script>

    <form id="f"> <!-- Formulario para introducir coordenadas y color del pixel -->
      x<input name="x" type="text" size="2" /> <!-- Campo texto para la abscisa -->
      y<input name="y" type="text" size="2" /> <!-- Campo texto para la ordenada -->
      color<input name="c" type="color" /> <!-- Indicar el color del punto -->
      <!-- Botón para llamar a la función que muestra el punto -->
      <input type="button" value="pintar" onclick="dibujar()" />
    </form>

    <canvas id="lienzo" width="400" height="400"> <!-- Lienzo para pintar el punto -->
    </canvas>

  </body>

</html>
```

---------- **pixelxy.js**

```
function ponerPixel(contexto, x, y, color){

  contexto.fillStyle = color; //configura el color para pintado

  //pintar un punto (debe ser 1x1, para mejorar su visualización es de 3x3)
  contexto.fillRect( x, y, 3, 3 );

}

function dibujar(){

  var form = document.getElementById("f"); //accedemos al formulario

  var x = parseInt(form.elements["x"].value); //obtenemos el valor de la abscisa

  var y = parseInt(form.elements["y"].value); //obtenemos el valor de la ordenada

  var c = form.elements["c"].value; //obtenemos el color

  var canvas = document.getElementById("lienzo"); //accedemos al lienzo de dibujo

  var contexto = canvas.getContext("2d"); //obtenemos el contexto 2d del lienzo

  ponerPixel(contexto, x, y, c); //pintamos el pixel

  contexto.fillText( "(" + x + "," + y + ")", x+4, y); //mostramos la coordenada

}
```

Especificar coordenadas con el clic del ratón. (Ejemplo 3)

El siguiente ejemplo muestra cómo trabajar con el evento onMouseDown asociado al lienzo de dibujo para indicar dónde queremos pintar el pixel. El resultado nos permitirá pintar un pixel y mostrar sus coordenadas cada que demos clic sobre el canvas.



---------- **pixelclic.html**

```
<!DOCTYPE html>
<html>

  <head>
    <title>Pintando pixeles con el clic del ratón</title>
  </head>

  <body>

    <!-- Archivo con la rutina para pintar el pixel utilizando javascript -->
    <script src="pixelclic.js"></script>

    <!-- Lienzo para pintar el punto al hacer clic con el ratón -->
    <canvas id="lienzo" width="400" height="400" onmousedown="dibujar(event)">
    </canvas>

  </body>

</html>
```

---------- **pixelclic.js**

```
function ponerPixel(contexto, x, y, color){

  contexto.fillStyle = color; //configura el color para pintado

  //pintar un punto (debe ser 1x1, para mejorar su visualización es de 3x3)
  contexto.fillRect( x, y, 3, 3 );

}

function dibujar(){

  var x = event.offsetX; //obtenemos la abscisa de la ubicación del clic del ratón

  var y = event.offsetY; //obtenemos la ordenada de la ubicación del clic del ratón

  var canvas = document.getElementById("lienzo"); //accedemos al lienzo de dibujo

  var contexto = canvas.getContext("2d"); //obtenemos el contexto 2d del lienzo

  ponerPixel(contexto, x, y, "#0000FF"); //pintamos el pixel x,y en color azul

  contexto.fillText( "(" + x + "," + y + ")", x+4, y); //mostramos la coordenada

}
```

Pintar puntos al azar sobre un canvas. (Ejemplo 4)

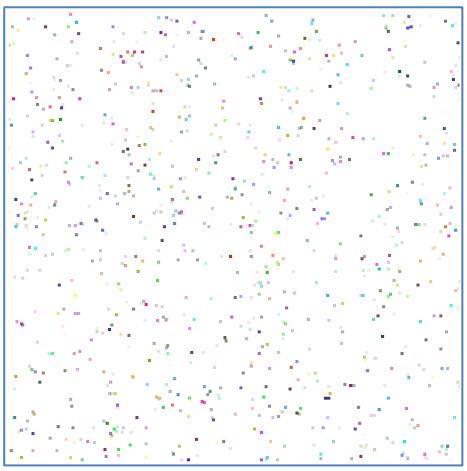
Solo como una manera de seguir practicando con el uso del canvas desde javascript, muestro el siguiente ejemplo que pinta 1000 pixeles al azar en el lienzo de dibujo:

---------- **pixeles.html**

```
<!DOCTYPE html>
<html>

  <head>
    <title>1000 puntos al azar</title>
  </head>

  <!-- Al cargar la página se llamara a la rutina para pintar -->
  <body onload="dibujar()">

    <!-- Archivo con la rutina para pintar los pixeles utilizando javascript -->
    <script src="pixeles.js"></script>

    <canvas id="lienzo" width="400" height="400"> <!-- Lienzo para pintar los puntos -->
    </canvas>
  </body>

</html>
```

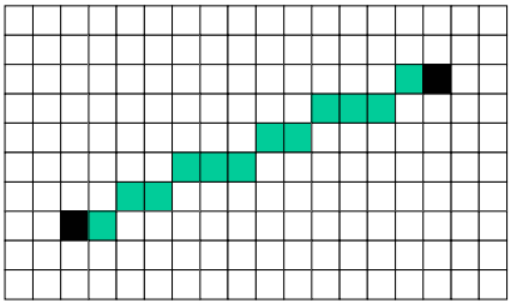
---------- **pixeles.js**

```
function ponerPixel(contexto, x, y, r, g, b, a){

  //configura el color para pintado
  contexto.fillStyle = "rgba("+r+","+g+","+b+","+a+")";

  //pintar un punto (debe ser 1x1, para mejorar su visualización es de 3x3)
  contexto.fillRect( x, y, 3, 3 );
}

function dibujar(){ //Esta función es llamada al cargar el documento html

  var canvas = document.getElementById("lienzo"); //accedemos al lienzo de dibujo

  var contexto = canvas.getContext("2d"); //obtenemos el contexto 2d del lienzo

  for (var i=0; i<1000; i++){ //pintaremos 1000 pixeles

    //obtenemos las coordenadas x,y al azar
    var x = Math.floor((Math.random() * 400) + 1);
    var y = Math.floor((Math.random() * 400) + 1);

    //obtenemos las intensidades de los canales RGBa del pixel al azar
    var r = Math.floor((Math.random() * 255) + 1); //componente roja entre 0 y 255
    var g = Math.floor((Math.random() * 255) + 1); //componente verde entre 0 y 255
    var b = Math.floor((Math.random() * 255) + 1); //componente azul entre 0 y 255
    var a = Math.random(); //la transparencia del pixel es un número real entre 0 y 1

    ponerPixel(contexto, x, y, r, g, b, a); //pintamos el pixel
  }
}
```



Primitiva Línea

Para dibujar líneas hay que hallar todos los puntos medios entre el inicial y el final. El problema es que la ubicación de cada pixel se representa con un entero y las posiciones halladas mediante la ecuación son valores reales. Por lo tanto, se necesitan formas eficientes de dibujar la recta lo más parecida posible a la realidad, a pesar de las limitaciones que las pantallas de píxeles nos pongan.

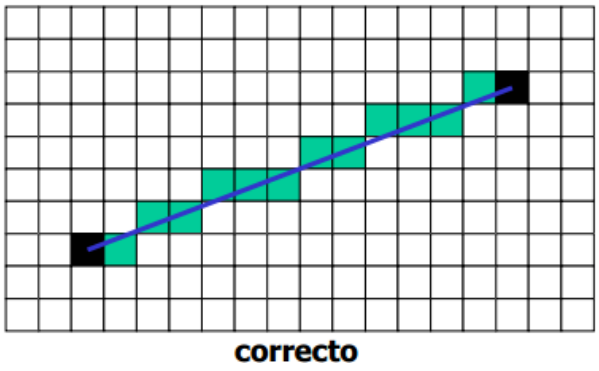


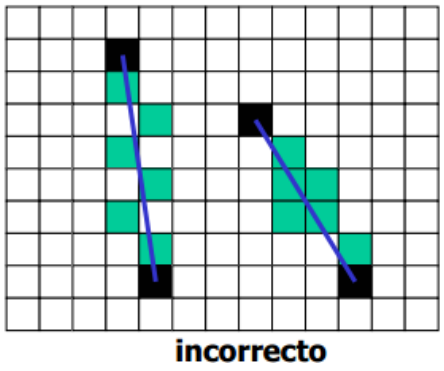
Consideraciones para el dibujo de rectas

Hay que calcular las coordenadas de los pixeles que estén lo más cerca posible de una línea recta ideal, infinitamente delgada, superpuesta sobre la matriz de pixeles.

Las consideraciones que un buen algoritmo debe cumplir son:

- La secuencia de pixeles debe ser lo más recta posible.
- Las líneas deben dibujarse con el mismo grosor e intensidad independientemente de su inclinación.
- Las líneas deben dibujarse lo más rápido posible.
- Las posiciones de los pixeles son valores enteros, y los puntos obtenidos de la ecuación de una recta son reales, lo que deriva en un error (aliasing). A menor resolución, mayor es el efecto.





La ecuación de la recta.

Dada una recta mediante un punto, P=(x_0, y_0), y una pendiente **m**, se puede obtener la ecuación de la recta a partir de la fórmula de la pendiente (ecuación punto-pendiente):

$$\mathbf{y - y_0 = m\,(x - x_0)}$$

donde **m** es la tangente del ángulo que forma la recta con el eje de abscisas **x**.

Ejemplo: La ecuación de la recta que pasa por el punto A = (2, -4) y que tiene una pendiente de m = $-\frac{1}{3}$ es: x + 3y + 10 = 0

Al sustituir los datos en la ecuación, resulta lo siguiente:

$y - y_0 = m\,(x - x_0)$
$y - (-4) = -\frac{1}{3}\,(x - 2)$
$3(y + 4) = -1\,(x - 2)$
$3y + 12 = -x + 2$
$x + 3y + 12 = 2$
$x + 3y + 10 = 0$

Forma simplificada de la ecuación de la recta

Si se conoce la pendiente m, y el punto donde la recta corta al eje de ordenadas es (0, b), podemos deducir, partiendo de la ecuación general de la recta, $y – y_0 = m\,(x - x_0)$:

$y - b = m\,(x - 0)$
$y - b = mx$
$\mathbf{y = mx + b}$

Las líneas rectas pueden ser expresadas mediante esta ecuación, donde **x**, **y** son variables en un plano cartesiano. En dicha expresión **m** es denominada la "pendiente de la recta" y está relacionada con la inclinación que toma la recta respecto a los ejes que definen el plano:

$$m = \frac{y_1 - y_0}{x_1 - x_0}.$$

Mientras que **b** es el denominado "término independiente" u "ordenada al origen" y es el valor del punto en el cual la recta corta al eje vertical en el plano.

Trazando una recta basándonos en un punto y su pendiente.

Jugar con la información anterior nos ayudara a diseñar e implementar un algoritmo que nos permita desplegar líneas rectas en la pantalla de una computadora.

Ejemplo 1: Trazar la recta que pasa por el punto A = (-6, -5) y que tiene una pendiente de m = $\frac{2}{3}$:

La pendiente **m** del ejemplo es positiva lo que significa que la recta se deberá graficar de abajo hacia arriba (para el eje y) y de izquierda a derecha (para el eje x).

La pendiente $\frac{2}{3}$ nos dice que para encontrar los puntos de la recta a graficar debemos movernos (partiendo del punto inicial, A) 2 unidades en el eje **y** por cada 3 unidades en **x**.

Los puntos obtenidos al aplicar esta afirmación son:

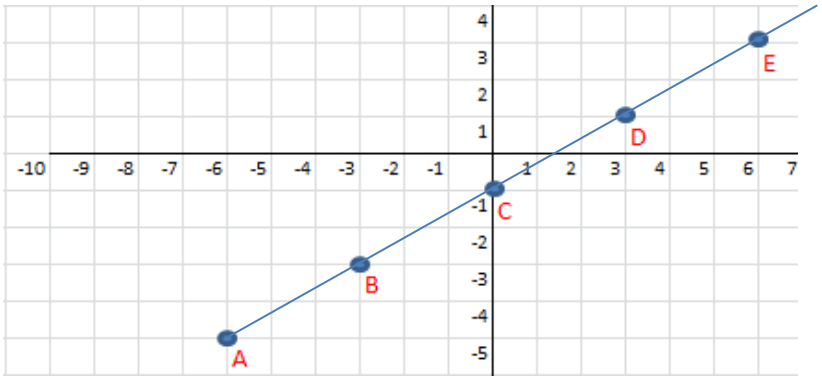
A = (-6, -5)

B = (-3, -3)

C = (0, -1)

D = (3, 1)

E = (6, 3)



Podemos ver en la gráfica que el punto C es en el que la recta corta al eje vertical en el plano; esto nos dice que **b** tiene un valor de **-1**.

Podríamos calcular el valor de b con los datos iniciales del ejemplo:

y = mx + b

-5 = $\frac{2}{3}$ (-6) + b

-5 = $\frac{2}{3}$ x $\frac{-6}{1}$ + b

-5 = $-\frac{12}{3}$ + b

-5 = - 4 + b
+4 +4

-1 = b

Con un par de puntos (A y E por ejemplo) podríamos calcular o comprobar la pendiente:

$m = \frac{y_1 - y_0}{x_1 - x_0}$

$m = \frac{3-(-5)}{6-(-6)}$

$= \frac{8}{12} = \frac{4}{6} = \frac{2}{3}$

Ejemplo 2: Trazar la recta que pasa por el punto A = (-8, 5) y que tiene una pendiente de m = $-\frac{1}{4}$:

La pendiente **m** del ejemplo es negativa lo que significa que la recta se deberá graficar de arriba hacia abajo (para el eje y) y de derecha a izquierda (para el eje x).

La pendiente $-\frac{1}{4}$ nos dice que para encontrar los puntos de la recta a graficar debemos movernos (partiendo del punto inicial, A) 1 unidad en el eje **y** por cada 4 unidades en **x**.

Los puntos obtenidos al aplicar esta afirmación son:

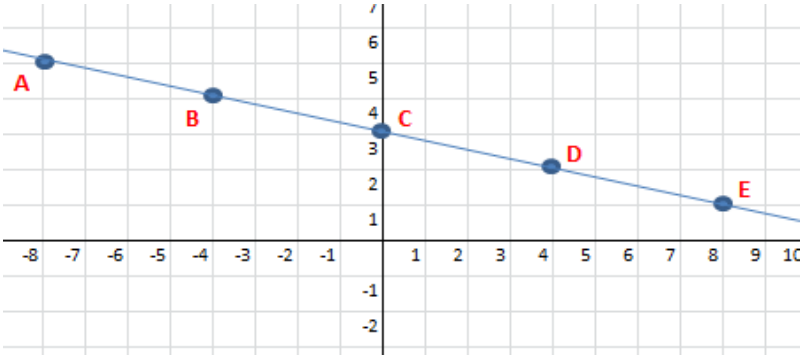
A = (-8, 5)

B = (-4, 4)

C = (0, 3)

D = (4, 2)

E = (8, 1)



Podemos ver en la gráfica que el punto C es en el que la recta corta al eje vertical en el plano; esto nos dice que **b** tiene un valor de **3**.

Podríamos calcular el valor de b con los datos iniciales del ejemplo:

y = mx + b

5 = $-\frac{1}{4}$ (-8) + b

5 = $-\frac{1}{4}$ x $\frac{-8}{1}$ + b

5 = $\frac{8}{4}$ + b

5 = 2 + b
-2 -2

3 = b

Con un par de puntos (A y E por ejemplo) podríamos calcular o comprobar la pendiente:

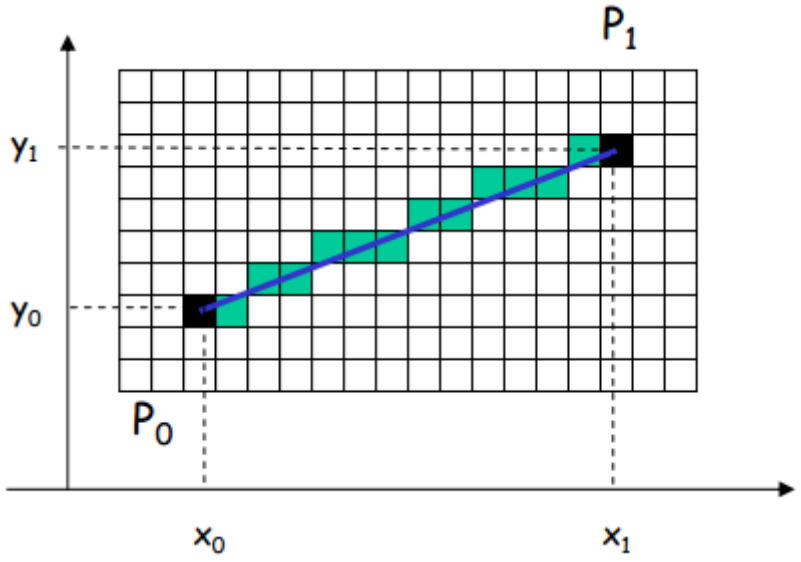
$m = \frac{y_1 - y_0}{x_1 - x_0}$

$m = \frac{1-5}{8-(-8)}$

$= \frac{-4}{16} = -\frac{2}{8} = -\frac{1}{4}$

Un algoritmo simple.

Tomando como base la ecuación de la recta (y = mx + b) podemos trazar una línea mediante un algoritmo en el que se especifiquen los puntos inicial (P0) y final (P1) del segmento que deseamos mostrar:



El algoritmo consistiría en:

Calcular $m = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0}$

Calcular $b = y_0 - mx_0$

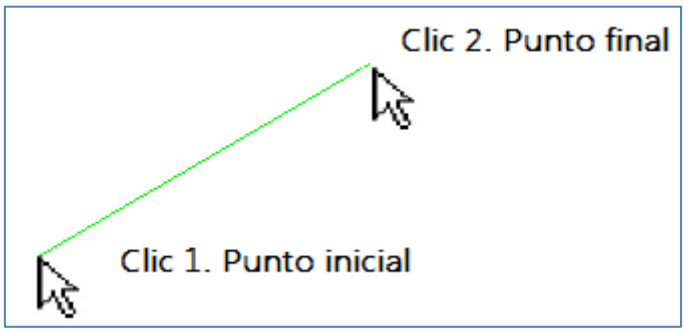
```
Para x=x0 hasta x=x1
   y = mx + b
   Pintar Pixel (x, round(y))
```

El algoritmo anterior presenta los siguientes inconvenientes:

- No es muy eficiente.
- Cada paso requiere una multiplicación flotante, una suma y un redondeo.

Implementación en javascript. (Ejemplo 5)

Para pintar la línea daremos dos clics en el lienzo de dibujo, uno para indicar el inicio de la línea y otro para indicar el fin de la misma. El algoritmo requiere que la línea tenga su punto inicial más a la izquierda que el punto final.



---------- **lineaec.html**

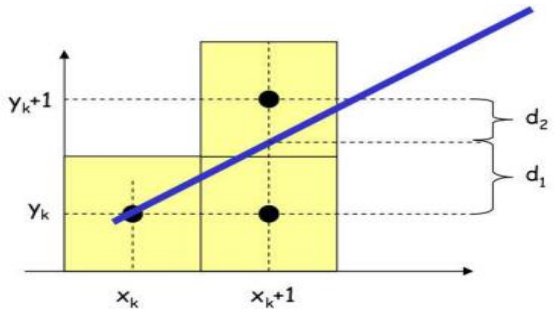
```
<!DOCTYPE html>
<html>

  <head>
    <title>Trazado de líneas basados en la ecuación de la recta</title>
  </head>

  <body>

    <!-- Archivo con la rutina para pintar las líneas utilizando javascript -->
    <script src="lineaec.js"></script>

    <!-- Lienzo para pintar las líneas -->
    <canvas id="lienzo" width="400" height="400" onmousedown="dibujar(event)">
    </canvas> <!-- Función dibujar se ejecuta cada que se hace clic sobre el lienzo -->

  </body>

</html>
```

---------- **lineaec.js**

```
//xi, yi se usan para guardar las coordenadas del primer punto de la recta
var xi=0;
var yi=0;

//la recta se define por dos puntos, el punto inicial de la recta
//será la posición donde se haga clic por primera vez y el punto
//final estara definido por la ubicación del segundo clic
var primerPunto=true; //bandera para controlar los clics

function ponerPixel(contexto, x,y, color){ //Pintar un punto

  contexto.fillStyle = color;

  contexto.fillRect(x, y, 1, 1);

}

//Para pintar una recta esta función deberá ser ejecutada dos veces
//la primera vez captura las coordenadas del punto incial de la recta
//y lo grafica sobre el lienzo. La segunda vez toma las coordenadas
//del segundo punto y pinta la línea llamando a la función lineaEcuacionRecta.
function dibujar(event) { //Se ejecuta cada que se hace clic sobre el lienzo

  var canvas = document.getElementById("lienzo"); //accedemos al lienzo de dibujo

  var contexto = canvas.getContext("2d"); //obtenemos el contexto 2d del lienzo

  if(primerPunto){ //Si es el primer clic, se lee el primer punto de la línea

    xi=event.offsetX; //Guardamos la abscisa

    yi=event.offsetY; //guardamos la ordenada

    ponerPixel(contexto, xi, yi, "#00FF00"); //ponemos el punto inicial en verde

  }else //Si es el segundo clic, pintamos la línea con los valores xi, yi
    //y la posición del último clic del ratón (event.offsetX, event.offsetY)
    lineaEcuacionRecta(xi, yi, event.offsetX, event.offsetY, contexto, "#00FF00");

  primerPunto =! primerPunto; //Invertir el valor de la bandera para pintar más líneas

}

function lineaEcuacionRecta(x0, y0, x1, y1, contexto, color){

  var m=(y1-y0)/(x1-x0); //calcular pendiente

  var b= y0 - (m*x0); //determinar el punto donde la recta se cruza con el eje y

  for(var x=x0; x<=x1; x++){ //El punto inicial debe estár más a la izq. q el final

    var y= (m*x)+b; //Ecuación de la recta

    ponerPixel(contexto, x, Math.round(y), color); //pintar el sig. punto de la línea

  }

}
```

Implementación del algoritmo de Bresenham para líneas. (Ejemplo 6)

El algoritmo calcula cuál de dos píxeles es el más cercano a la trayectoria de una línea. El pixel (Xk, Yk) se divide en (Xk+1, Yk) y (Xk+1, Yk+1) y hay que decidir cuál pintar, calculando la distancia vertical entre el centro de cada pixel y la línea real.




```
Si 0<|m|<1
  *Se capturan los extremos de la línea y se almacena el extremo izquierdo en (x0,y0).
  *Se carga (x0,y0) en el búfer de estructura (se traza el primer punto)
  *Se calculan las constantes Δx,Δy, 2Δy y 2Δy-Δx y se obtiene el valor inicial para el
    parámetro de decisión p0=2Δy-Δx.
  Para j=0 mientras j<Δx
  *En cada xk a lo largo de la línea, que inicia en k=0 se efectúa la prueba siguiente:
      Si pk<0
          *Trazamos (xk+1,yk).
          *Asignamos pk+1= pk+2Δy.
      Sino
          *Trazamos (xk+1,yk+1).
          *Asignamos pk+1= pk+2Δy-2Δx.
  Fin Para
Si |m|>1
   *Recorremos la dirección en pasos unitarios y calculamos los valores sucesivos
     de x que se aproximen más a la trayectoria de la línea.
```

---------- **lineabresenham.html**

```
<!DOCTYPE html>
<html>

  <head>

    <title>Implementación del algoritmo de trazado de líneas de Bresenham</title>

  </head>

  <body>

    <!-- Archivo con la rutina para pintar las líneas utilizando javascript -->
    <script src="lineabresenham.js"></script>

    <!-- Lienzo para pintar las líneas -->
    <canvas id="lienzo" width="400" height="400" onmousedown="dibujar(event)">
    </canvas> <!-- La función dibujar se ejecuta en cada clic sobre el lienzo -->

  </body>

</html>
```

---------- **lineabresenham.js**

```
//xi, yi se usan para guardar las coordenadas del primer punto de la recta
var xi=0;
var yi=0;

//la recta se define por dos puntos, el punto inicial de la recta
//será la posición donde se haga clic por primera vez y el punto
//final estara definido por la ubicación del segundo clic
var primerPunto=true; //bandera para controlar los clics

function ponerPixel(contexto, x,y, color){
  contexto.fillStyle = color;
  contexto.fillRect(x, y, 1, 1);
}

//Para pintar una recta esta función deberá ser ejecutada dos veces
//la primera vez captura las coordenadas del punto incial de la recta
// y lo grafica sobre el lienzo. La segunda vez toma las coordenadas
//del segundo punto y pinta la línea llamando a la función lineaBresenham.
function dibujar(event){ //Esta función se ejecuta cada que se hace clic sobre el lienzo

  var canvas = document.getElementById("lienzo"); //accedemos al lienzo de dibujo

  var contexto = canvas.getContext("2d"); //obtenemos el contexto 2d del lienzo

  if(primerPunto){ //Si es el primer clic, se lee el primer punto de la línea

    xi=event.offsetX; //Guardamos la abscisa
    yi=event.offsetY; //guardamos la ordenada

    ponerPixel(contexto, xi, yi, "#FF00FF"); //ponemos el pixel en rosa
  }else //Si es el segundo clic, pintamos la línea con los valores xi, yi
  //y la posición del último clic del ratón (event.offsetX, event.offsetY)
    lineaBresenham(xi, yi, event.offsetX, event.offsetY, contexto, "#FF00FF");

  primerPunto =! primerPunto;

}

//Implementación del algoritmo de Bresenham para líneas
function lineaBresenham(x0, y0, x1, y1, contexto, color){

  var dx = Math.abs(x1-x0);
  var dy = Math.abs(y1-y0);

  var sx = (x0 < x1) ? 1 : -1;
  var sy = (y0 < y1) ? 1 : -1;

  var err = dx-dy;

  while(x0!=x1 || y0!=y1){

    ponerPixel(contexto, x0, y0, color);

    var e2 = 2*err;

    if (e2 >-dy){ err -= dy; x0 += sx; }

    if (e2 < dx){ err += dx; y0 += sy; }
  }
}
```

Partimos de que las coordenadas de los pixeles en una imagen son coordenadas enteras y que conocemos los extremos del segmento que forma la línea siendo sus coordenadas (x0, y0) y (x1, y1).
El algoritmo de Bresenham selecciona el entero 'y' correspondiente al pixel central que está más cercano del que sería calculado con fracciones y lo mismo para la coordenada 'x'. En las sucesivas columnas la coordenada 'y' puede permanecer con el mismo valor o incrementarse en cada paso a una unidad.

La ecuación general de la línea que pasa por los extremos conocidos es:

$$\frac{y - y_0}{y_1 - y_0} = \frac{x - x_0}{x_1 - x_0}$$

Puesto que conocemos la columna, 'x', la fila 'y' del pixel se calcula redondeando esta cantidad al entero más cercano según la siguiente fórmula:

$$y = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) + y_0$$

La pendiente (y1- y0) / (x1- x0) depende sólo de las coordenadas de los extremos y puede ser previamente calculada, y el valor ideal de 'y' para los sucesivos valores enteros de 'x' se puede calcular a partir de y0 e ir añadiendo en varias ocasiones la pendiente.

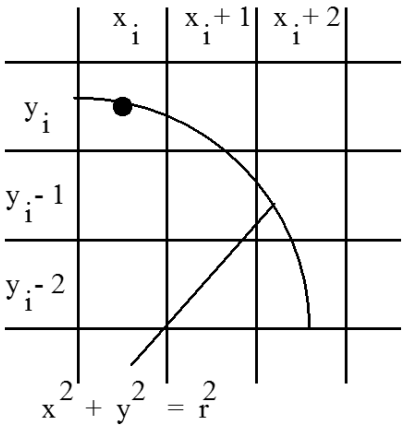
Implementación del algoritmo de Bresenham para círculos. (Ejemplo 7)

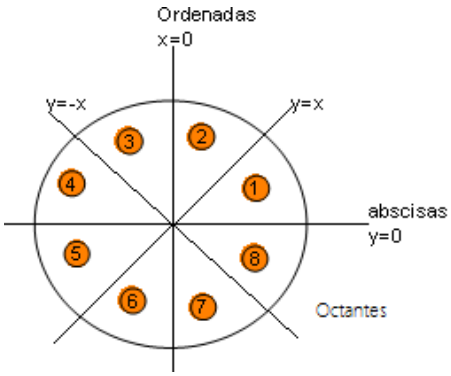
En graficación, el algoritmo de "midpoint circle" es un algoritmo utilizado para determinar los puntos necesarios para dibujar un círculo. El algoritmo es una variante del algoritmo de la línea Bresenham, por lo que es a veces conocido como algoritmo de círculo Bresenham, aunque en realidad no fue inventado por él.

Se basa en el empleo de la función implícita: $F(x, y)=x^2 + y^2 - R^2 = 0$

- Si F(x, y) = 0 el punto está en la curva del círculo
- Si F(x, y) > 0 el punto está encima de la curva
- Si F(x, y) < 0 el punto está debajo de la curva

Hacemos el análisis para un octante de 0 a x=y, y hacemos simetrías de ocho puntos.



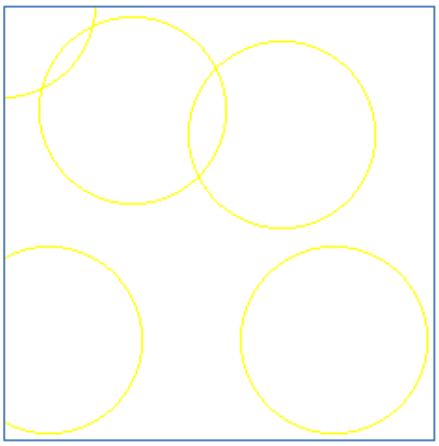


---------- **circulo.html**

```
<!DOCTYPE html>
<html>

  <head>
    <title>Implementación del algoritmo de trazado de círculos</title>
  </head>

  <body>

    <!-- Archivo con la rutina para pintar círculos utilizando javascript -->
    <script src="circulo.js"></script>

    <!-- Lienzo para pintar los círculos -->
    <canvas id="lienzo" width="400" height="400" onmousedown="dibujar(event)">
    </canvas> <!-- La función dibujar se ejecuta con cada  clic sobre el lienzo -->

  </body>

</html>
```

---------- **circulo.js**

```
function ponerPixel(contexto, x,y, color){

  contexto.fillStyle = color;

  contexto.fillRect(x, y, 1, 1);

}

function dibujar(event){

  var canvas = document.getElementById("lienzo");

  var contexto = canvas.getContext("2d");

  //círculo amarillo con radio de 50 pixeles.
  //cuyo centro está definido por las coordenadas del clic del ratón
  circulo(event.offsetX, event.offsetY, 50, contexto, "#FFFF00");

}

//Implementación del algoritmo de Bresenham para circulos
function circulo(xc, yc, radio, contexto, color){

  var x = radio*(-1);

  var y = 0;

  var r=radio;

  var err = 2-2*radio;

  do{

    ponerPixel(contexto, xc-x, yc+y, color);

    ponerPixel(contexto, xc-y, yc-x, color);

    ponerPixel(contexto, xc+x, yc-y, color);

    ponerPixel(contexto, xc+y, yc+x, color);

    r=err;

    if(r>x) err+=++x*2+1; if(r<=y) err+=++y*2+1;

  }while(x<0);

}
```

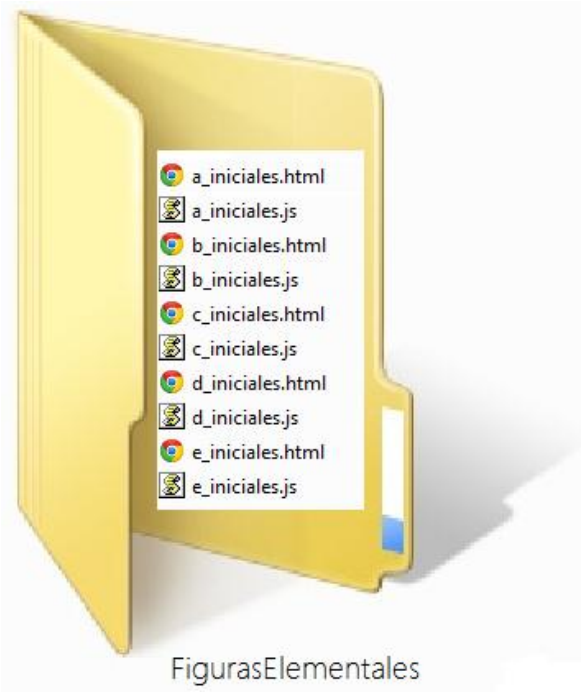


El ejemplo anterior permite dibujar círculos amarillos cuyo centro es el clic del ratón y con un radio de 50 pixeles.

Ejercicios.

Los siguientes ejercicios se componen de dos archivos cada uno, un documento html con la interfaz de usuario y otro con la lógica de la solución al ejercicio, de la misma manera en la que aparecen los ejemplos anteriores. En la interfaz de usuario (el archivo html) deberán agregar sus datos (Nombre del alumno, Nombre del profesor, Titulo del ejercicio o descripción, Universidad, Carrera, Materia y Ciclo escolar).

Deberán publicarse en un servicio de webhosting. Se integraran en una carpeta y formarse de la siguiente manera: (Reemplace la palabra "iniciales", por sus propias iniciales)



a) El ejemplo 3, permite graficar puntos al hacer clic con el ratón. Todos los puntos graficados aparecen en color azul. Modifique los archivos **pixelclic.html** y **pixelclic.js** para que se permita especificar al usuario el color del punto a graficar. No incluya un botón, sólo el selector con la paleta de colores html.

b) El ejemplo 4 muestra 1000 pixeles en colores y posiciones al azar dentro del lienzo de dibujo. Recordemos que en los ejemplos se utiliza un lienzo de 400x400 pixeles. Modifique los archivos **pixeles.html** y **pixeles.js** para que todos los pixeles del lienzo se iluminen con un color al azar.

c) El ejemplo 5 permite al usuario dibujar una línea recta basados en la ecuación de la recta especificando los puntos de inicio y fin de la línea mediante un par de clics del ratón. El ejemplo requiere que el punto inicial de la línea este más a la izquierda que el punto final, ya que es solo un algoritmo demostrativo. Modifique los archivos **lineaec.html** y **lineaec.js** para que grafique líneas sin importar dónde estén ubicados los puntos que la definen. Para este ejemplo, basta con intercambiar los puntos inicial y final dentro de la función lineaEcuacionRecta() si el inicial está a la derecha del final. Lo anterior, antes de iniciar con el cálculo de los puntos medios.

d) El ejemplo 6 implementa el algoritmo de Bresenham para líneas. Permite pintar una línea especificando los puntos de inicio y fin de la línea mediante un par de clics del ratón. Las líneas graficadas se presentan siempre en color rosa. Modifique los archivos **lineabresenham.html** y **lineabresenham.js** para incorporar un selector de color para que la línea se pinte del color indicado por el usuario. Agregue también el código necesario para que se muestren en texto las coordenadas de los puntos inicial y final de la línea. El ejemplo 6 obliga al usuario a indicar los puntos inicial y fina de cada línea, modifíquelo también para que una vez introducida la primer línea, el punto final de esta sea el punto inicial de la siguiente, así con cada clic se pintará una línea desde donde termina la anterior hasta la posición del último clic dado.

e) El ejemplo 7 implementa el algoritmo del punto medio para círculos. Muestra círculos de 100 pixeles de diámetro tomando como centro el clic del ratón. Modifique los archivos **circulo.html** y **circulo.html** para incorporar un selector utilizado por el usuario para indicar el color del círculo. Agregue también una caja de texto donde pueda especificar el radio del mismo.