Curso de Gráficos

Por Steven R. Davidson

http://graficos.conclase.net

Prefacio

Bienvenidos/as al Curso de Programación de Gráficos. Con este curso intento explicar la **teoría** acerca de la creación de gráficos en pantalla. Comenzaremos con crear imágenes en 2 dimensiones (2D), en los primeros capítulos. Luego, pasaremos a dar la teoría de modelar objetos en 3 dimensiones (3D) y su representación como una imagen en 2D: la pantalla. Huelga decir que el autor no lo sabe todo, por lo que quizá no se den todos los temas que algunas personas quisieran. Se tratará de dar la mayoría de los temas, pero es posible que algunos sean demasiados avanzados o novedosos. El campo de crear gráficos se ramifica en otros campos de la ciencia y el estudio; por ejemplo,

- Animación
- Arte
- Física
- Fotografía
- Matemáticas
- Modelado de sistemas virtuales
- Psicología
- Vídeo
- Visualización

Este curso está diseñado para programadores de C/C++ con conocimientos avanzados o experiencia. Los temas requeridos incluyen:

- Bases de Datos: manejo de ficheros y organización de datos.
- Listas dinámicas enlazadas: árboles, colas, pilas, etc..
- Matemáticas pre-universitarias: ecuaciones/funciones, geometría, trigonometría, vectores, etc..

Es preferible que el lector tenga conocimientos de alguna librería o API gráfica para el sistema operativo o entorno que se esté usando. El curso presentará y usará como base el API de MS-Windows® para el sistema operativo MS-Windows® de Microsoft™. El autor ha incluido un "paquete" de códigos fuente y ficheros de cabecera para aquellas personas que no sepan usar el API de MS-Windows®. Cada paquete irá incrementando de contenido a medida que vayamos dando más temática. El paquete sirve como base y **no** como sustitución del API de MS-Windows®. Esto quiere decirse que el paquete contendrá lo mínimo para poder empezar a practicar, pero no incluirá todo lo tratado en cada capítulo.

El autor quiere dejar MUY claramente que este curso NO tratará de un tutorial para usar ninguna librería ni API específicas. El curso trata de la teoría de la programación de gráficos.

Esto no implica que no vaya a haber ejemplos ni usos prácticos ni ejercicios. De hecho, los primeros capítulos se dedicarán a ejemplos prácticos para que el curso coja soltura y no sea tan *pesado* con la teoría.

Capítulo 1 - Conceptos

Antes de dar paso al curso, se presentarán y se definirán varios términos relacionados con el tema de gráficos.

Sistema visual u ocular. Las imágenes que percibimos son debidas a nuestro sistema visual: ojo, cerebro, y nervio óptico, principalmente. El ojo, que forma parte de este sistema visual, se caracteriza por sus componentes oculares. El iris controla la cantidad de luz que es permitida entrar al ojo. La lente ocular concentra la luz permitida en el ojo para formar una imagen. La proyección de la imagen es situada en la retina - una estructura de dos dimensiones. La retina se compone principalmente de conos y bastoncillos, que son células especiales. Estos conos y bastoncillos tienen la funcionalidad de recibir estímulos de luz para nuestra visión diurna y nocturna, respectivamente. La luz es descompuesta en las intensidades que corresponden a los colores rojo, verde, y azul. Este trío de colores se denomina los colores primarios. A partir de los colores primarios podemos conseguir otros tres colores: cian, amarillo, y magenta, llamados los colores complementarios. Cian, amarillo, y magenta son complementarios a los colores rojo, verde, y azul, respectivamente, ya que combinados formarán el color blanco.

Resolución gráfica. En un sistema gráfico, se representan colores para formar una imagen. Los colores son combinados para formar tonos y otros colores en un área pequeña. Este área es llamada "píxel", que viene del inglés: *picture element* o elemento pictográfico (o de imagen). Este elemento o píxel es la parte fundamental más pequeña que contiene información para crear una imagen en un sistema gráfico. Un conjunto de píxeles forma una malla para crear una imagen completa. La mayoría de los sistemas gráficos (por no decir todos) usa una malla rectangular con las mismas dimensiones de una superficie rectangular: anchura y altura. La anchura es el número de columnas y la altura, el número de filas de tal superficie gráfica. El número de píxeles en la anchura y en la altura constituyen la resolución (gráfica) de la imagen. Por ejemplo, 640 x 480 significa que la resolución gráfica de la imagen constituye 640 columnas y 480 filas de píxeles formando una malla de 307.200 píxeles. Analizando otro ejemplo: 800 x 600, comprobamos que obtenemos una imagen de 480.000 píxeles. Esta imagen contiene muchos más píxeles y por tanto mayor resolución pictográfica. Con una mayor resolución gráfica se obtiene una mejor calidad de imagen.

Cuando hablamos de mejor resolución gráfica, hablamos de usar un número mayor de líneas de píxeles en la horizontal y vertical. Realmente, lo que está pasando es que cuanto más se junten los píxeles, nos es más difícil distinguir entre uno y otro píxel vecino. Esto es debido en parte a la distancia física entre los conos en nuestra retina y en parte a las condiciones visuales que dependen de nuestro entorno. Esta habilidad de discernir detalles se llama *acuidad* o *agudeza* (visual). Nuestra agudeza es menor por el desgaste, al envejecer, y por efectos de nuestro entorno; por ejemplo, al disminuir el contraste y brillo.

La resolución y el concepto de una malla de áreas de colores son análogos a un mosaico. En tal forma de arte, un mosaico intenta representar una imagen a partir de azulejos pequeños de determinados colores y tonos. Cada azulejo intenta aproximarse en color a un área de la imagen, combinando los colores en susodicho área, como refleja la siguiente imagen:

Modelos de colores. En sistemas gráficos, los colores son descompuestos y representados por combinaciones de valores numéricos. Generalmente, se habla de un color diferente para cada combinación de valores. En realidad, se está hablando de tonos diferentes de colores, pero en el argot informático cada tono es tratado como un color diferente. Por ejemplo, si se habla de un sistema gráfico de 15 bits por píxel (bpp) tenemos a nuestra disposición 32.768 colores, aunque en verdad sean unos cuantos colores y varios miles de tonos. De igual forma, podemos manipular 24 bpp obteniendo un conjunto de 16.777.216 colores.

Existen varios modelos de descomposición de colores, especialmente usados en tecnología. Los modelos

más populares y usados son RGB/A, CMYK, HLS, e YUV:

- **RGB**. Estas siglas provienen del inglés *red*, *green*, y *blue*; esto es, *rojo*, *verde*, y *azul*, respectivamente. Este trío de colores intenta modelar el sistema visual humano. La mayoría de los monitores y televisores se basan en este modelo, debido a que están basados en la emisión de luz. Este modelo también se le atribuye la propiedad de *sistema aditivo*, ya que se añaden las intensidades para formar un color.

- **RGBA**. Se trata del mismo modelo RGB, pero con otra propiedad: canal *alfa*. Este canal se usa como un índice de la transparencia en un píxel. Esto nos sirve a la hora de mezclar varios colores designados para un solo píxel. Este modelo es más reciente y se suele usar para crear efectos y técnicas visuales como "anti-aliasing", niebla, llamas, y objetos semi-transparentes: cristal, agua, vidrieras, etcétera.

- **CMYK**. Las siglas representan, en inglés, *cyan*, *magenta*, *yellow*, y *black*; esto es, *cian*, *magenta*, *amarillo*, y *negro*, respectivamente. Los tres primeros colores son complementarios a los de RGB. Este modelo se usa más en la imprenta para crear imágenes a partir de tintas de colores. Para obtener un color más vivo, se le añade la tinta negra también. Este modelo tiene la propiedad de *sistema sustractivo*, ya que se restan las intensidades a la luz. El pigmento se obtiene porque el material absorbe la energía de la luz y por tanto una parte de su espectro. La energía que transmite se percibe como un color determinado. Por ejemplo, nosotros percibimos el color verde, porque tal pigmento absorbe todos los "colores" del espectro de la luz emitida y transmite/refleja el color verde.

- **HLS**. En inglés, las siglas son *hue*, *lightness* o *luminance*, y *saturation*; esto viene a ser, *matiz*, *brillo*, y *saturación*, respectivamente. Este modelo se basa en lo empírico de nuestra percepción visual. En lugar de usar un modelo tricolor para formar otros colores y tonos, el modelo HLS se basa en tres propiedades que sirven para definir los colores que percibimos. El matiz es el color que solemos denominar: azul, violeta, rojo, dorado, etcétera. El brillo describe la vividez y brillo de un tono de color. La saturación diferencia la percepción de un tono "puro" a otro tono que ha sido mezclado con blanco para formar un tono pastel - suave. Este modelo es usado por artistas, prinicipalmente.

- **YUV** también escrito **YCbCr** o incluso **YPbPr**. Este modelo se basa en el modelo RGB, pero restringiendo y descomponiendo algunos valores del RGB. YUV se usa en señales de televisión y en equipos de vídeo y grabación como S-Vídeo, y MPEG-2 y por tanto DVD. Originalmente, la principal razón de usar este modelo fue para mantener compatibilidad con los televisores analógicos en blanco y negro, cuando se introdujeron los televisores en color.

  El componente Y es el brillo total. U y V son componentes cromáticos (colores) basados en los componentes rojo y azul del modelo RGB. El proceso comienza con una imagen en blanco y negro (Y). Luego, se obtiene el componente U, mediante una resta entre Y y el componente azul del RGB original, y V mediante una resta entre Y y el componente rojo.

Representación gráfica. Nuestro objetivo como programadores gráficos y como artistas es la representación de nuestra realidad. Debemos aproximarnos a la imagen ideal usando varias técnicas visuales. En definitiva, estamos tratando de engañar a nuestro sistema visual para que nuestra imagen aparente ser real. Dicho de otra forma, tenemos que visualizar una imagen de resolución infinita en un

área de resolución limitada. Éste es el concepto principal de la representación gráfica. Todas nuestras técnicas visuales y manipulaciones de una o varias imágenes se basan en presentar una imagen engañando al usuario lo suficientemente como para que la imagen final aparente ser el objeto real. Sin embargo, a veces debemos hacer sacrificios en nuestra "misión" de crear ilusiones. Ya que existen limitaciones según el hardware de nuestro sistema gráfico, es posible que decidamos reducir la resolución por falta de memoria, tiempo de computación, falta de tiempo y recursos, etcétera. Tal filosofía se puede ilustrar por el cuadro de Michell Foucault, *Ceci n'est pas une pipe* (1968).

El título, "Esto no es una pipa", es cierto, ya que es una *representación* de una pipa, y no una pipa de verdad.

Modelar. Huelga decir que nuestra herramienta para crear gráficos es al fin y al cabo nuestro ordenador (o computadora). Esto implica que tendremos que trabajar con valores numéricos y por consiguiente deberemos realizar cálculos para conseguir nuestra imagen. Los valores numéricos que usaremos representan vértices, líneas, vectores, vórtices (vértices en tres dimensiones), y demás conceptos matemáticos, al igual que la manipulación de bits para colores, combinación de imágenes, etcétera. Siguiendo la misma lógica de la representación gráfica, debemos representar un objeto usando estos conceptos matemáticos. Este concepto de modelado se basa en la composición de objetos primitivos como vértices y líneas. Por ejemplo,
Si queremos representar un cuadrado, tendremos que guardar información acerca de los vértices y líneas para poder recrear la imagen de un cuadrado. Dibujamos una línea desde un vértice a otro representando un lado de este cuadrado. Luego, dibujamos una segunda línea desde el último vértice a uno nuevo; y así

sucesivamente hasta dibujar los cuatro lados del cuadrado. Acabamos de definir, en términos de objetos primitivos, el modelo de un cuadrado. Del mismo modo, podemos definir las características de un cubo: un conjunto de 8 vértices, 4 aristas, y 6 caras o planos. Con esta información, podemos crear una imagen desde cualquier punto de vista: de lado, desde arriba, abajo, etcétera. También podemos girar el objeto, mudarlo, cambiarlo de tamaño, y deformarlo, con tan sólo aplicar fórmulas a la información definida. Al tratar con modelos de objetos de dos o tres dimensiones (o más), nos aproximamos a un nivel más matemático, y por tanto, el ordenador nos ayuda a realizar los cálculos necesarios.

Mapear. Ya que estamos usando conjuntos de píxeles para representar imágenes, a veces necesitamos que la imagen represente dimensiones más "humanas". Digamos que queremos crear una imagen de un mapa. Requerimos que nuestro mapa nos dé información precisa. Por ello, cada línea en nuestro mapa debe representar una longitud determinada; por ejemplo, cada tramo de carretera equivale a 10 kilómetros. Si esto es cierto para todos los tramos de nuestra imagen, entonces nuestro mapa está a escala. Análogamente, podemos tener varias imágenes pero de distintos tamaños. Necesitamos combinar estas imágenes para que tengan la misma escala. Esto se denomina mapear, del inglés *mapping*, o cambiar de escala.

El cambio de escala también puede producirse debido al modelo matemático. Por ejemplo, si queremos representar una ecuación matemática: $y = 2x + 3$, podemos calcular los valores de y a partir de valores escogidos de x. Estas coordenadas se usarán para poder seleccionar un píxel que las represente. Sin embargo, no todos los valores van a ser posibles representar, según los valores escogidos. Realmente estamos hablando de representar una imagen en dimensiones matemáticas en un sistema gráfico de píxeles. Debemos realizar un cambio de escala y seguramente de coordenadas.

El mapeo es necesario al tratar con objetos de diferentes dimensiones y orientaciones. Se debe transformar el estado de todos los objetos a una base para que sean compatibles. En el ejemplo de la ecuación matemática, acabaríamos transformando las coordenadas cartesianas calculadas a coordenadas en píxeles.

Capítulo 2 - Trazar Ecuaciones

Para comenzar, veamos un ejemplo práctico del uso de gráficos: trazar la gráfica de una ecuación. En este capítulo trazaremos la ecuación: $y = 3x^2 - 6$. Como ya sabemos, esta ecuación describe una parábola cóncava (hacia "arriba") - en el sentido positivo por el eje-Y. Nosotros, como matemáticos, trazamos tal gráfica tomando ciertos valores de ***x*** para calcular valores de ***y*** y así crear coordenadas por donde pasará la gráfica. Al ser humanos, no nos gusta tomar todos los posibles valores, por lo que creamos una tabla de valores calculando algunas coordenadas, hasta que tengamos una idea de dónde situar la gráfica. En nuestra ecuación, sabemos de antemano que se trata de una parábola, por lo que ya conocemos su forma y estructura. En papel, aproximamos el trazado a la gráfica real, ya que no nos hace falta tanta precisión; o sea, lo hacemos a ojo de buen cubero. Sin embargo, esto no es posible al crear la gráfica en pantalla: necesitamos ser precisos.

Cambio de Coordenadas I

La ventaja que tenemos es que el ordenador no se "cansará" al calcular todos los valores que requerimos, por lo que no tendremos problemas de precisión. Por otro lado, estamos intentado representar una imagen de dimensiones infinitas - el plano cartesiano, debido a los valores de ***x*** que son infinitos, en un área de dimensiones finitas - la pantalla. Por lo tanto, debemos representar la gráfica ajustándonos a las dimensiones de nuestra pantalla. Las dimensiones de la imagen equivalen a la resolución gráfica establecida. Para este ejemplo, usaremos una resolución de 800x600.

Ahora tenemos que pensar qué representa cada píxel que activemos - demos color. Si establecemos que cada píxel equivale a una unidad matemática, entonces no obtendremos una imagen presentable. Hay que tener en cuenta que las unidades matemáticas no tienen por qué ser valores enteros, mientras que los píxeles sí deben serlos. Lo que tenemos que hacer es decidir los valores mínimos y máximos a usarse en la ecuación. Digamos que queremos ver parte de la gráfica usando los valores de ***x*** : **[-3, +3]**; o sea, $x_{ui} = -3$ y $x_{uf} = 3$, los cuales representan los valores mínimos y máximos de las unidades del plano cartesiano, respectivamente. Aún así, no podemos usar todos los valores en este conjunto, ya que serían infinitos. Como ya sabemos, no podemos representar valores infinitos en un sistema de valores finitos (limitados). Tenemos que repartir estos valores cartesianos de entre los posibles valores de la pantalla del mismo eje X; es decir, tenemos que conseguir cambiar el intervalo **[-3, +3]** al de **[0, 799]**. Los valores iniciales y finales son fáciles de averiguar: $x_{ui} = -3 \Rightarrow 0$ y $x_{uf} = 3 \Rightarrow 799$. Los demás valores deberán ser calculados:
Primeramente, debemos cambiar la escala o longitud: 6 (=3-(-3)) unidades cartesianas a 800 píxeles (el número de columnas); esto implica que existen 6/800 unidades cartesianas por cada píxel, que es lo mismo que decir: 0,0075 unidades/píxel. Dicho de otro modo, cada píxel representará 0,0075 unidades cartesianas. Miremos unos cuantos valores, para ilustrar este concepto:

Valores de X

Unidades Cartesianas	Píxeles
-3,0000	0

-2,9925	1
-2,9850	2
-2,9775	3
.	.
.	.
.	.
2,9700	796
2,9775	797
2,9850	798
2,9925	799

También podemos establecer la relación realizando la operación inversa: 800/6 = 133,3333 píxeles/ unidad. Aquí podemos ver que sería algo difícil representar 133,3333 píxeles, ya que los píxeles son siempre enteros. Esto implica que obtendremos errores al aproximarnos a un número entero; en este caso, 133 (~133,3333).

Cual sea la relación, podemos averiguar un píxel determinado a partir de una coordenada cartesiana determinada, y viceversa. Por ejemplo, si tenemos la coordenada-X de un píxel, x_p = 345, para poder averiguar el valor de x en unidades cartesianas, realizamos la siguiente operación:

$$\frac{6 \text{ unidades}}{800 \text{ píxeles}} = \frac{|x_u - x_{ui}| \text{ unidades}}{|345 - x_{pi}| \text{ píxeles}} \Rightarrow$$

$$|345 - 0| \text{ píxeles} * 0{,}0075 \text{ unidades/píxel} = |x - (-3)| \text{ unidades} \Rightarrow$$

$$x = -0{,}4125 \text{ unidades.}$$

Por otro lado, podemos averiguar el píxel correspondiente a la coordenada-X, x_u = -2,0000, aplicando la simple regla de tres:

$$\frac{800 \text{ píxeles}}{6 \text{ unidades}} = \frac{|x_p - x_{pi}| \text{ píxeles}}{|-2{,}0000 - x_{ui}| \text{ unidades}} \Rightarrow$$

$$1{,}000 \text{ unidad} * 133{,}3333 \text{ píxeles/unidad} = |x_p - 0| \text{ píxeles} \Rightarrow$$

$$x = 133{,}3333 \text{ píxeles} \Rightarrow x = 133 \text{ píxeles.}$$

Del mismo modo, tenemos que averiguar los valores de y en unidades cartesianas correspondientes con los valores en píxeles. El número de filas es 600 píxeles, según nuestra resolución que hemos elegido. Ya que los valores del eje-Y son imaginarios, éstos pueden ser calculados. Por esta razón, el conjunto de valores puede ser ajustado según los valores inicial y final del eje-X. Con $x_{ui} = -3$, obtenemos $y_u = 3\,(x_{ui})^2 - 6 \Rightarrow y_u = 21$. Con $x_{uf} = 3$, obtenemos $y_u = 21$. Ahora averiguaremos la coordenada del valor mínimo de y_u de la curva con la siguiente fórmula: $6x_u = 0 \Rightarrow x_u = 0$, y por tanto, $y_u = -6$. La fórmula usada proviene de la derivada de nuestra ecuación: $y_u = 3\,(x_u)^2 - 6$, para averiguar el punto crítico.

Ahora tenemos que los valores de ***y*** se encuentran entre **[-6, +21]**; esto es, $y_{ui} = -6$ e $y_{uf} = +21$. Por lo tanto, necesitamos realizar otro cambio de escala: 27 (=21-(-6)) unidades cartesianas a 600 píxeles (el número de filas). Esto quiere decirse que tenemos 27/600 = 0,0450 unidades/píxel. Veamos unos cuantos valores:

Valores de Y

Unidades Cartesianas	**Píxeles**
-6,0000	0
-5,9550	1
-5,9100	2
-5,8650	3
.	.
.	.
.	.
20,8200	596
20,8650	597
20,9100	598
20,9550	599

Calculemos el píxel que corresponda al valor en unidades cartesianas de $y_u = 3\ (-2{,}000)^2 - 6 \Rightarrow y_u = 6{,}000$. Nuevamente se trata de aplicar la regla de tres con la información que tenemos:

$$\frac{600 \text{ píxeles}}{27 \text{ unidades}} = \frac{|y_p - y_{pi}| \text{ píxeles}}{|6{,}0000 - y_{ui}| \text{ unidades}} \Rightarrow$$

$$|6{,}0000 - (-6)| \text{ unidades} * 22{,}2222 \text{ píxeles/unidad} = |y_p - 0| \text{ píxeles} \Rightarrow$$

$$y = 266{,}6666 \text{ píxeles} \Rightarrow y = 267 \text{ píxeles.}$$

Por lo tanto, la coordenada (-2, 6), en el plano cartesiano, es representada por la pareja columna-fila (133, 267), en la pantalla.

Observaciones

Es posible que el lector haya hecho unos cuantos cálculos u observaciones y haya descubierto que,
a) Aunque tengamos una serie de píxeles con coordenadas continuas a lo largo del eje-X, las coordenadas del eje-Y no son contiguas, en su mayor parte. Esto depende de la gráfica y, por tanto, del tipo de ecuación. Si estuviéramos trazando la gráfica de una ecuación lineal de poca pendiente: $y = mx + b$, $m : [0,1]$, donde m es la pendiente, entonces todos los valores de cada coordenada (x_p, y_p) serían contiguos. Perdemos precisión en la coordenada-Y, al ser precisos en la coordenada-X, y viceversa. Un

arreglo que podemos hacer es trazar líneas continuas entre cada píxel que calculemos. Esto implica que cada píxel se convertirá en un vértice para luego ser unidos con líneas. En resumen, aproximamos la curva de la gráfica usando líneas. Esto queda mejor explicado en la siguiente sección: *Explicación Detallada*.

b) Las fórmulas presentadas no sirven para hallar los valores finales de la coordenada ***x*** ni ***y***. En nuestro ejemplo, estos valores corresponderían a: $x_{uf} = 3$ e $y_{uf} = 21$; sus homólogos en píxeles son: $x_{pf} = 800$ e $y_{pf} = 600$. Claro está, estos valores son 1 píxel más allá de la resolución en cada dirección. Esto es debido a que estamos aproximando un intervalo de valores (cartesianos) como un valor único (píxel). Por ejemplo, el valor $x_p = 4$ representa el conjunto de valores x_u : [-2,9700, -2,9625). Si nos fijamos, el valor limitante, -2,9625, no pertenece a este intervalo, pero todos los valores anteriores pertenecientes sí son aproximados por $x_p = 4$. Cuando llegamos al último intervalo, x_u : [2,9925, 3,0000), notamos que el último valor $x_{uf} = 3$ no forma parte de este intervalo, aunque por defecto tiene la misma representación. No podemos hacer mucho para corregir esta inexactitud; simplemente se nos han acabado las columnas. Sin embargo, podemos hacer una de estas dos posiblidades para intentar arreglarlo:
1. Trazar una línea usando como la última coordenada, $x_{pf} = 800$ (el siguiente valor). La mayoría de los sistemas gráficos permitirán dibujar en zonas inexistentes ya que éstos realizan la tarea de eliminar píxeles "sobrantes" o "irrepresentables". Con esto, conseguiremos la parte de la línea en el eje-Y. Aunque no lleguemos justamente a la coordenada x_{pf}, llegaremos lo suficientemente cerca.

2. Invertir el proceso de trazado. En vez de averiguar el valor de ***y*** a partir del valor de ***x*** y posteriormente obtener su correspondencia en píxeles, trataremos de averiguar los valores de ***y*** que pertenezcan a sus complementos de ***x***.

Explicación Detallada

Como ya se ha explicado, los píxeles representan intervalos de valores. En nuestro ejemplo, $x_p = 133$ representa los valores de x_u : [-2,0025, -1,9950), calculando el valor de y_u obtenemos $y_u = 6,03001875$ que corresponde a $y_p = 267,33375 = 267$. Realmente, el píxel (133,267) representa un área de valores: x_u : [-2,0025, -1,9950) e y_u : [6,0150, 6,0600). Esto se refleja en la *Figura 1*.

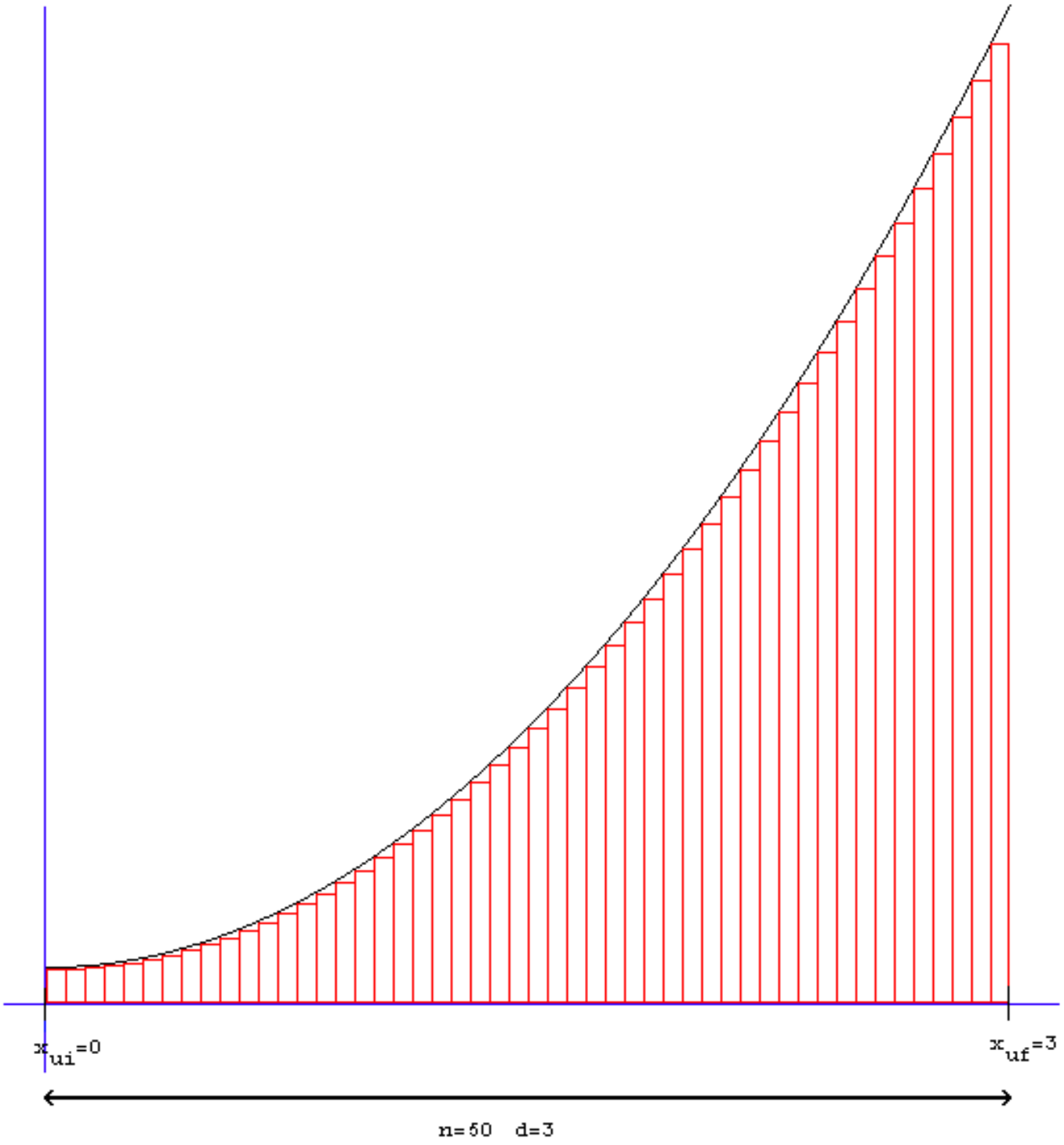


Figura 1

En la *Figura 1*, cada rectángulo representa la coordenada x_p del píxel. La anchura de cada rectángulo indica el recorrido de valores de x_u que representa. Del mismo modo, actúa cada rectángulo para la coordenada y_p. Esto se muestra claramente en la *Figura 2*:

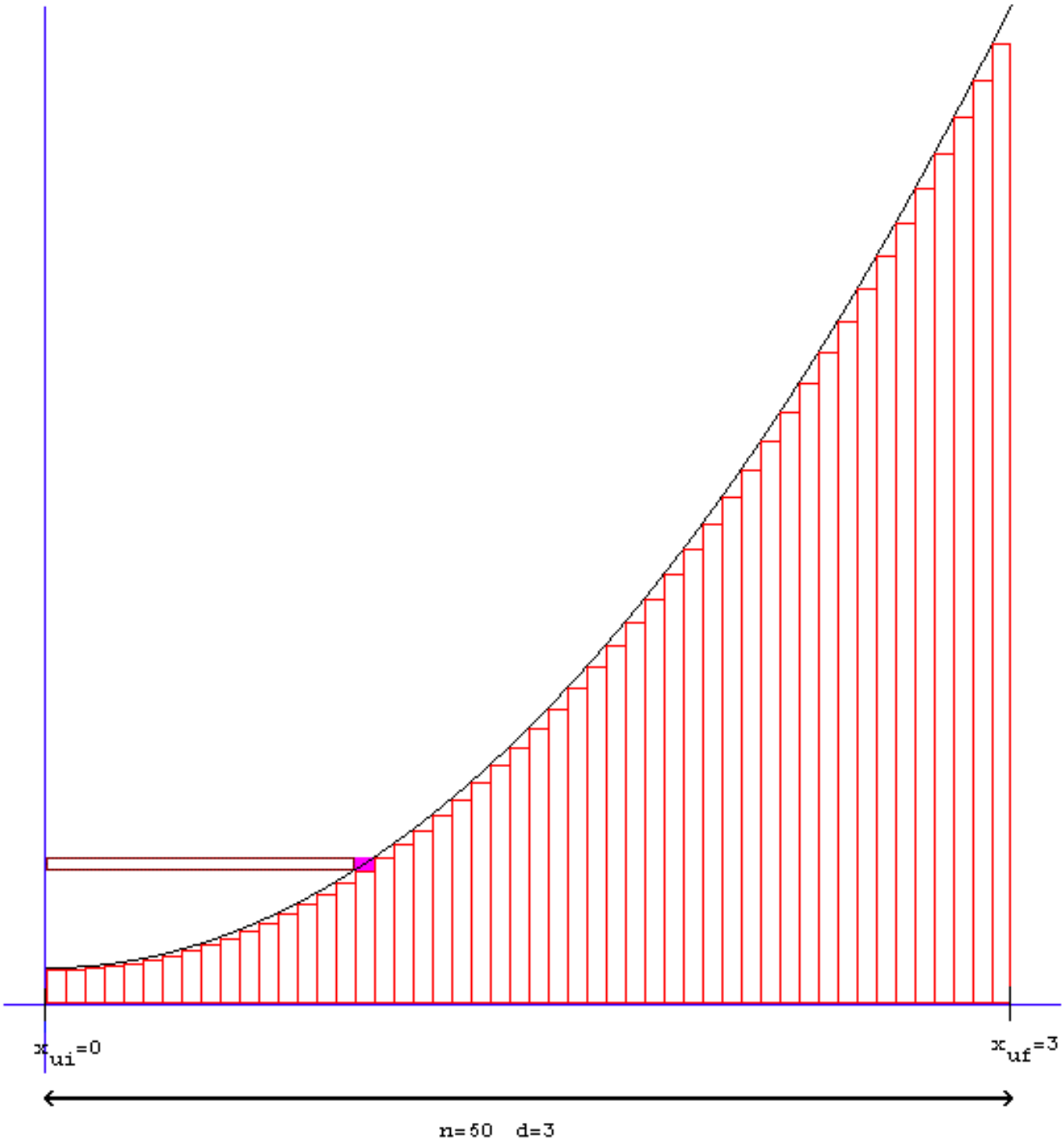


Figura 2

El área de color violeta, representa el píxel (x_p,y_p) para los recorridos de los valores de x_u e y_u.

Para aquellos lectores que tengan conocimientos de cálculo diferencial, ambas figuras asemejan la aproximación del área debajo de una curva. Al área real (precisa) la llamamos una integral. Sin embargo, nosotros sólo calculamos los valores para crear una serie y no una suma, como es el método de la suma de Riemann. En la *Figura 2*, tenemos los datos suficientes para comprobar nuestro método:
n : Número de particiones = rectángulos,

d : Distancia de nuestro intervalo.
x_{ui} : Valor inicial de nuestro intervalo.

Usando los valores en la gráfica, obtenemos:
$x_0 = [x_{ui}, x_{ui} + 3/50)$
$x_1 = [x_{ui} + 3/50, x_{ui} + 6/50)$
$x_2 = [x_{ui} + 6/50, x_{ui} + 9/50)$
...
$x_a = [x_{ui} + a*d / n, x_{ui} + (a+1)*d / n),\ \ a = 0,1,2,...,n-1$

a representa el número de columnas de nuestra resolución gráfica. Así es como elegimos a nuestros valores en la coordenada ***x***. Para los valores en la coordenada ***y***, aplicamos nuestra ecuación teniendo en cuenta el número de particiones a lo largo del eje-Y. Por esta razón debemos realizar un cambio de coordenadas del plano cartesiano al plano gráfico en píxeles.

Al ver ambas figuras, podemos observar que tal método nos es suficiente para coordenadas que no se distancien mucho entre sí. Observando la curva hacia la derecha, nos damos cuenta de que los valores de cada partición se alejan bastante, y por tanto optamos por trazar líneas entre cada píxel. Esto es análogo al método trapezoidal para la aproximación de un área debajo de una curva. Es decir, en vez de usar rectángulos, usamos trapezoides.

La otra observación de ambas figuras es la representación del último valor x_{uf}. La curva continúa, pero la partición no puede representar x_{uf}. Una alternativa es invertir el proceso de aproximación. En vez de usar el método de rectángulos inscritos, usar rectángulos circunscritos o exteriores en vez de interiores. Generalmente no podemos decidir cuál método nos saldrá más rentable, ya que depende de la ecuación que queramos representar y por tanto de la forma de la gráfica.

Otro método que podemos implementar es realizar los cálculos inversos: a partir de coordenadas en píxeles, averiguar si pertenecen a la ecuación en unidades cartesianas. Sin embargo, para usar este método, tendríamos que comprobar todos los píxeles en la pantalla. Para crear imágenes de fractales, sí haremos uso de este método. Para algunos fractales, la imagen no es el trazado de la ecuación, sino que se **basa** en una ecuación. Este caso lo veremos más detenidamente en el siguiente capítulo: Fractales.

Cambio de Coordenadas II

Aún no hemos acabado con nuestra gráfica. Al calcular los valores y cambios de escala, lo hemos hecho en base a una condición: la orientación del sistema de coordenadas del plano cartesiano sea la misma que el sistema gráfico; en nuestro caso, es la pantalla. Generalmente, los sistemas gráficos comienzan por la coordenada (0,0) en la esquina superior izquierda. Esto implica que la orientación en el eje-X es positiva de izquierda a derecha y en el eje-Y es positiva de arriba a abajo. En el plano cartesiano, por regla común, la orientación positiva en el eje-X es de izquierda a derecha y en el eje-Y de abajo a arriba. Como podemos observar, la orientación en el eje-X es la misma que en el plano cartesiano: de izquierda a derecha, pero no en el eje-Y. Debido a esta diferencia, debemos asegurarnos de la orientación; de lo contrario, veremos nuestros trazados en el sentido inverso; como si fueran vistos en un espejo.

Para arreglar las orientaciones, necesitamos arreglar nuestras fórmulas. Como ya se ha dicho, la orientación en el eje-X no necesita cambios, pero el eje-Y, sí los necesita. Usaremos el mismo ejemplo, y_u = 6,0000. El píxel y_p en su orientación correcta se calcula de la siguiente forma:

$$\frac{600 \text{ píxeles}}{27 \text{ unidades}} = \frac{|(y_{pi} + altura_p) - y_p| \text{ píxeles}}{(6{,}0000 - y_{ui}) \text{ unidades}} \Rightarrow$$

$$|6{,}0000 - (-6)| \text{ unidades} * 22{,}2222 \text{ píxeles/unidad} = |(0+600) - y_p| \text{ píxeles} \Rightarrow$$

$$y_p = 333{,}3333 \text{ píxeles} \Rightarrow y_p = 333 \text{ píxeles.}$$

altura$_p$: es la altura o el número de filas de nuestra resolución. Aplicando un poco de lógica y álgebra, podemos usar la siguiente fórmula:

$$\frac{600 \text{ píxeles}}{27 \text{ unidades}} = \frac{|(y_{pf} + 1) - y_p| \text{ píxeles}}{|6{,}0000 - y_{ui}| \text{ unidades}} \Rightarrow$$

$$|6{,}0000 - (-6)| \text{ unidades} * 22{,}2222 \text{ píxeles/unidad} = |(599+1) - y_p| \text{ píxeles} \Rightarrow$$

$$y_p = 333{,}3333 \text{ píxeles} \Rightarrow y_p = 333 \text{ píxeles.}$$

Al final, el píxel (133, 333) representa correctamente el punto (-2, 6), en la mayoría de los sistemas gráficos.

Fórmulas

Descripción de Variables

x_{pi}	Valor inicial de la coordenada X en píxeles
y_{pi}	Valor inicial de la coordenada Y en píxeles
x_{pf}	Valor final de la coordenada X en píxeles
y_{pf}	Valor final de la coordenada Y en píxeles
x_p	Valor arbitrario de la coordenada X en píxeles
y_p	Valor arbitrario de la coordenada Y en píxeles
x_{ui}	Valor inicial de la coordenada X en unidades cartesianas
y_{ui}	Valor inicial de la coordenada Y en unidades cartesianas

x_{uf}	Valor final de la coordenada X en unidades cartesianas
y_{uf}	Valor final de la coordenada Y en unidades cartesianas
x_u	Valor arbitrario de la coordenada X en unidades cartesianas
y_u	Valor arbitrario de la coordenada Y en unidades cartesianas
$anchura_p$ $\|x_{pf} - x_{pi} + 1\|$	Número de columnas a representar en píxeles
$altura_p$ $\|y_{pf} - y_{pi} + 1\|$	Número de filas a representar en píxeles
$anchura_u$ $\|x_{uf} - x_{ui}\|$	Distancia entre los valores final e inicial de las coordenadas X en unidades cartesianas
$altura_u$ $\|y_{uf} - y_{ui}\|$	Distancia entre los valores final e inicial de las coordenadas Y en unidades cartesianas

Fórmulas Usadas

$$\frac{anchura_u}{anchura_p} = \frac{|x_u - x_{ui}|}{|x_p - x_{pi}|}$$

Para hallar el valor de x_u conociendo el valor de x_p

$$\frac{altura_p}{altura_u} = \frac{|(y_{pf} - y_{pi} + 1) - y_p|}{|y_u - y_{ui}|}$$

Para hallar el valor de y_p conociendo el valor de y_u si la orientación en píxeles es de sentido contrario a la del plano cartesiano

Algoritmo

Aquí exponemos el algoritmo para el trazado de una ecuación:

1. **Inicializar valores para: x_{ui}, x_{uf}, y_{ui}, y_{ui}, x_{pi}, x_{pf}, y_{pi}, y_{pf}**
2. **$anchura_u$ <- $|x_{uf} - x_{ui}|$**
3. **$altura_u$ <- $|y_{uf} - y_{ui}|$**
4. **$anchura_p$ <- $|x_{pf} - x_{pi} + 1|$**
5. **$altura_p$ <- $|y_{pf} - y_{pi} + 1|$**

```
6.  dx_up <- anchura_u / anchura_p
7.  dy_pu <- altura_p / altura_u
8.  x_u <- x_ui
9.  y_u <- función( x_u )
10. x_orig_p <- x_pi
11. y_orig_p <- y_pf + 1 - dy_up * |y_u - y_ui|
12. Bucle: x_p <- x_pi+1 hasta x_pf con incremento de 1
13.    x_u <- x_u + dx_up
14.    y_u <- función( x_u )
15.    y_p <- y_pf + 1 - dy_up * |y_u - y_ui|
16.    Dibujar_Línea( x_orig_p, y_orig_p, x_p, y_p )
17.    x_orig_p <- x_p
18.    y_orig_p <- y_p
```

función() es la ecuación cuya gráfica queremos trazar. La función *Dibujar_Linea()* hace alusión a la función básica de cualquier API o librería gráfica, para trazar una línea recta desde una coordenada de píxeles a otra.

Ejercicios

Enlace al paquete de este capítulo.

1. Escribir, para cada ecuación, un programa que trace las gráficas de las siguientes ecuaciones y según las restricciones dadas:
a) $\mathbf{y}(\mathbf{x}) = 0{,}5\mathbf{x}^4 - 2\mathbf{x}^3 + 0{,}33\mathbf{x}^2 - 0{,}75\mathbf{x} + 1$,
Restricciones: $x = [-1, +2]$, $y = [\mathbf{y}(2), \mathbf{y}(-1)]$,
Resolución: 100 x 100.

b) $\mathbf{y}(\mathbf{x}) = (1{,}5\mathbf{x}^3 - 2{,}6\mathbf{x}^2 + 3{,}3\mathbf{x} - 10) / (\mathbf{x}^2 - 1{,}5\mathbf{x} + 0{,}5)$,
Restricciones: $x = [0, +2]$, y = elige un intervalo apropiado,
Resolución: 300 x 300,
Observación: la ecuación contiene dos asíntotas verticales, x_1 y x_2. Calcula, con anterioridad, estos valores de x para que la ecuación no sea indeterminada en $\mathbf{x} = x_1$ ni en $\mathbf{x} = x_2$.
(Pista: Precalcula, $\mathbf{x}^2 - 1{,}5\mathbf{x} + 0{,}5 = 0$, ya que si el denominador es 0, entonces la división queda indeterminada y por tanto la ecuación se "dispara" a infinito en el eje-Y positivo).

c) Si $x < 1$, entonces $\mathbf{y}(\mathbf{x}) = 1 - \mathbf{x}^2$, Si $1 \le x \le 9$, entonces $\mathbf{y}(\mathbf{x}) = \ln \mathbf{x}$, Si $x > 9$, entonces $\mathbf{y}(\mathbf{x}) = \cos \mathbf{x}$,
Restricciones: $x = [-1, +12]$, y = elige un intervalo apropiado,
Resolución: 600 x 600.

d) $\mathbf{y}(\mathbf{x}) = |6 - \mathbf{x}^2|$,
Restricciones: $x = [-5, +5]$, $y = [0, +6]$,
Resolución: 300 x 300 para la gráfica, pero centrado en una resolución total de la ventana, portal, o

zona de dibujo de 500 x 500. Es decir, la gráfica debe dibujarse en unas dimensiones de 300 x 300, pero las dimensiones de la ventana o zona de dibujo son de 500 x 500. La gráfica debe estar centrada con respecto a la ventana.

2. Elige dos de las ecuaciones anteriores y reescribe sus programas para que trace las curvas usando el método descrito en el último párrafo de la sección *Observaciones*, bajo b)2.. Esto es, trazar la ecuación a partir de valores dados para y_p en vez de x_p. Compara las gráficas obtenidas con este método junto con el usado en el ejercicio anterior.

3. Escribir un programa para trazar la siguiente ecuación:
f(**x**) = 4 / π * [∑(sen(n**x**) / n)], donde *n* = 1,3,5,7,..., +∞ (infinito).
Si escribimos unos cuantos términos de la suma, obtenemos lo siguiente:
f(**x**) = 4 / π * [sen(**x**) + sen(3**x**) / 3 + sen(5**x**) / 5 + sen(7**x**) / 7 + sen(9**x**) / 9 + sen(11**x**) / 11 + sen(13**x**) / 13 + ...] ,
Restricciones: *x* = [-2π, +2π], *y* = [-1,5, +1,5],
Resolución: 800 x 800,
Como no podemos llegar hasta +*infinito*, entonces elegiremos un intervalo *finito*. Esto es, *n* := [1,N].
Para el valor de N, usa:
a) N = 3,
b) N = 7,
c) N = 21.

Observaciones: Cuanto mayor sea el valor de N, obtendremos una mayor precisión de la ecuación. Esto implica que nuestro error se reducirá a 0 (cero) o al menos un valor insignificante.

4. La ecuación presentada en el ejercicio #3 es, en realidad, la serie de Fourier para esta ecuación:
Si -π < *x* < 0, entonces **f**(**x**) = -1, Si 0 < *x* < +π, entonces **f**(**x**) = 1.
Reescribe el programa del ejercicio anterior #3 - usando las mismas restricciones y resolución - para dibujar la función anterior, presentada en este ejercicio #4. Si sabes manipular colores, usa distintos colores para cada trazado. Si no puedes distinguir un trazado del otro, entonces crea otro programa diferente y ejecuta ambos simultáneamente. Compara ambos trazados. Después de varias sumas (por ejemplo, N = 21), el trazado de la serie de Fourier (ejercicio #3) se parecerá o igualará al de esta ecuación (ejercicio #4).

Observaciones: 1) La función no define algunos valores de *f(x)* para todos los valores de *x*. El dominio (valores de *x*) es el intervalo (-π, 0) • (0, +π), • significa unión. Los valores x_1 = -π, x_2 = 0, y x_3 = +π no están definidos para *f(x)*. 2) La ecuación anterior debe ser trazada periódicamente. Es decir, el mismo trazado debe ser igual para los intervalos *x* := ... • (-3π, -2π) • (-2π, -π) • (-π, 0) • (0, +π) • (+π, +2π) • (+2π, +3π) • .... La periodicidad es de 2π; o sea, cada 2π unidades, la gráfica se repite.

5. Otra forma de crear curvas es basándonos en un parámetro común para *x* e *y*. Esto sería, **x** = **x**(**t**) e **y** = **y**(**t**), donde *t* es nuestro parámetro. Tales ecuaciones están en forma paramétrica. Escribir un programa para trazar las siguientes curvas paramétricas:
a) **x**(**t**) = sen(**t**), **y**(**t**) = cos(**t**), donde 0 ≤ *t* ≤ 2π,
b) **x**(**t**) = **t***cos(**t**), **y**(**t**) = 3**t***sen(**t**), donde 0 ≤ *t* ≤ 8π,
c) **x**(**t**) = (1 - **t**^{2}) / (1 + **t**^{2}), **y**(**t**) = 2**t** / (1 + **t**^{2}), donde -π ≤ *t* ≤ +π,
d) **x**(**t**) = a*cos(**t**) + b*cos(a**t**/2), **y**(**t**) = a*sen(**t**) - b*sen(a**t**/2), donde 0 ≤ *t* ≤ 2π,

Esta curva se denomina hipotrocoide. Prueba con unos cuantos valores para *a* y *b* y con diferentes intervalos de *t*; por ejemplo,

1) a = 8, b = 5
2) a = 6, b = 5
3) a = 6, b = 2
4) a = 12, b = 8
5) a = 14, b = 8,6

Elige las restricciones y la resolución que más convengan para cada caso.

Observaciones: Al usar ecuaciones paramétricas, el parámetro *t* sirve para calcular la pareja de valores (*x*, *y*) que realmente son (*x(t)*, *y(t)*) en el plano cartesiano. Es decir, *t* no es representado en el trazado de tales ecuaciones, sino que sirve de "ayudante" para calcular los posibles valores de *x* e *y*. En los anteriores ejercicios, nos basábamos en los valores incrementales de x_p. Sin embargo, ahora tenemos un valor en unidades cartesianas, *t*, que regula la iteración. El incremento de los valores de *t* no son fácilmente visibles. Intenta seguir esta fórmula para definir el incremento de *t* para cada iteración:
$delta\_t = |t_f - t_i| / n$,
donde t_i y t_f son los valores inicial y final de *t*, respectivamente. *n* es el número de muestras o valores de *t*, esto implica que cuanto mayor sea *n*, más coordenadas tendremos y por lo tanto más precisos serán nuestros cálculos. Esta precisión conlleva a una curva más curva (valga la redundancia).

Capítulo 3 - Fractales

Continuemos con otro ejemplo. En este capítulo veremos algo de teoría acerca de fractales, y cómo crear la imagen de un fractal a partir de su ecuación. Las aplicaciones de las imágenes de fractales no son muy populares, pero sí tienen un toque artístico y estético para algunas personas. Desde nuestro punto de vista, es un buen ejemplo para manipular los píxeles de la pantalla individualmente y también para tratar vectores, como un adelanto al tema.

Concepto

El término *fractal* viene de "dimensión fraccional" - en inglés, **fract**ion**al** dimension. La definición de un fractal es un objeto o cantidad que muestra auto-semejanza para todas las escalas. El objeto no tiene por qué demostrar la misma estructura en todas las escalas, pero sí el mismo tipo de estructura. Para que los lectores se hagan una idea, piensen que la longitud de un fractal es la longitud del borde (o la "costa") medida con reglas de diferentes longitudes. Cuanto más corta sea la regla, más grande es la longitud medida. Esta conclusión es conocida como la paradoja de la costa.

Las imágenes de fractales pueden ser generadas a partir de una definición recursiva. Tal definición puede ser el trazado de líneas rectas basándose en una gramática. Este conjunto de reglas se denomina *reglas de producción*, como es el caso de la curva de Koch, en la *Figura 1*.

Figura 1

También podemos dibujar el conjunto de valores que forman parte de una definición recursiva, como es el caso del fractal de Mandelbrot, en la *Figura 2*.

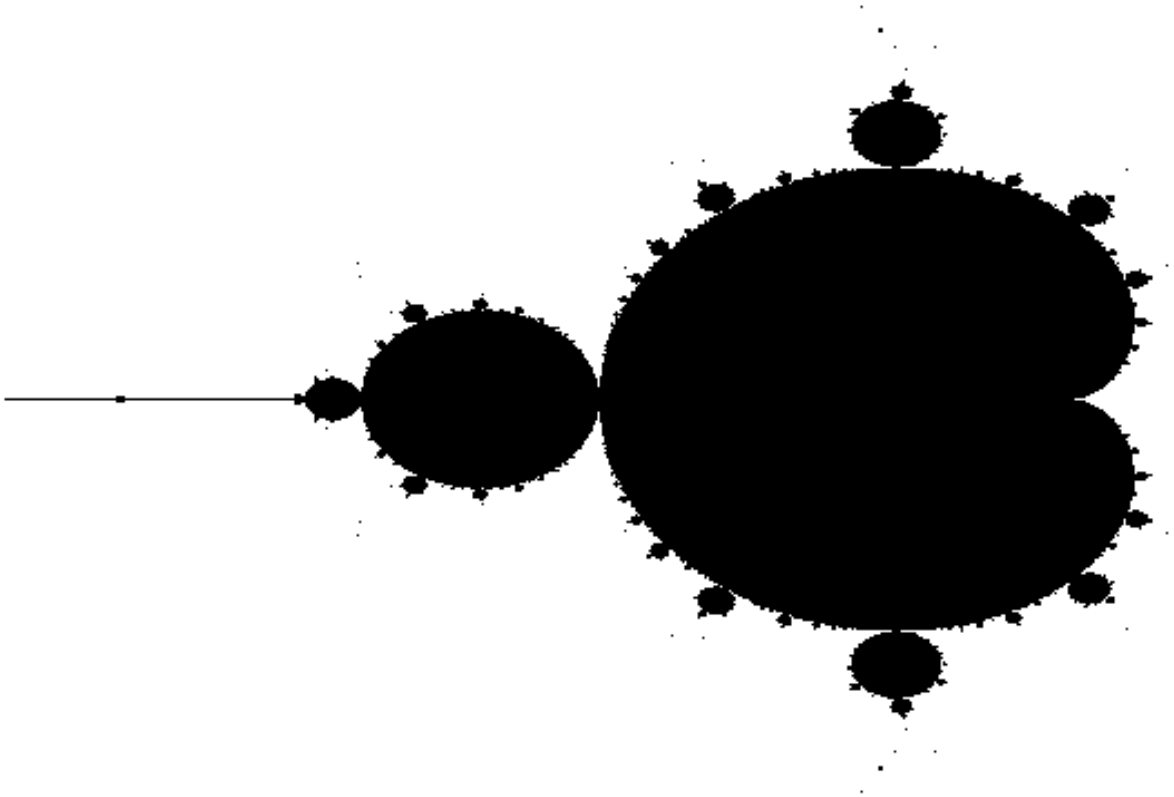


Figura 2

Explicaremos estos fractales en mayor detalle a continuación.

Capítulo 3 - Fractales: Curva de Koch

La curva de Koch se construye dividiendo un segmento en tres partes iguales. El segundo segmento - el segmento del medio - forma la base de un triángulo equilátero, pero sin representar este segmento - la base. Esto significa que tenemos un segmento horizontal seguido de dos segmentos, estos dos últimos en forma de triángulo, y luego seguido de un tercer segmento horizontal. Esta estructura es la primitiva para construir esta curva. En siguientes repeticiones, cada segmento de esta curva primitiva es a su vez sometido al mismo algoritmo: segmento horizontal seguido de dos segmentos terminando en la "punta" de un triángulo equilátero y seguido por un tercer segmento horizontal.

La estructura primitiva se puede definir usando la siguiente regla o axioma de producción:
A -> AIADDAIA,
donde A indica *avanzar*, lo cual implica *trazar* una línea recta,
I es girar a la *izquierda*, y
D es girar a la *derecha*.

En el caso de la curva de Koch, cada giro se hace con un ángulo de 60° = π/3 radianes. Los ángulos son 60° porque

la curva de Koch se construye en base a un triángulo equilátero. Todos los ángulos de un triángulo equilátero son 60°.

Podemos observar el estado inicial de la curva de Koch, que simplemente es una línea recta, en la *Figura 3*. Siguiendo la regla que tenemos, avanzamos una sola vez.

Figura 3

Para la primera iteración, obtenemos la secuencia: AIADDAIA con un ángulo de giro de 60°. Realmente estamos sustituyendo la regla de avanzar, inicialmente establecida, por la secuencia descrita por la regla de producción:

1. Avanzar el primer tercio.
2. Girar 60° a la izquierda.
3. Avanzar otro tercio.
4. Girar 60° a la derecha.
5. Girar 60° a la derecha.
6. Avanzar otro tercio.
7. Girar 60° a la izquierda.
8. Avanzar el último tercio.

Obtenemos una imagen como en la *Figura 4*.

Figura 4

En la segunda iteración, realizamos el mismo método basándonos en nuestra regla de producción. Aplicamos: A -> AIADDAIA para cada 'A'. Esto implica que sustituimos cada 'A' por su definición para lograr el siguiente producto: AIADDAIA I AIADDAIA DD AIADDAIA I AIADDAIA. He aquí el elemento recursivo. Al seguir esta regla obtenemos una imagen como la presentada en la *Figura 5*.

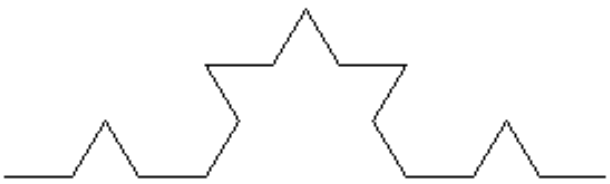


Figura 5

Podríamos reescribir la definición de esta regla para reflejar la recursividad:
A_0 -> Avanzar
A_{n+1} -> $A_nIA_nDDA_nIA_n$

donde *n* indica el número de repeticiones.

Para trazar estas líneas y seguir la regla de producción, nos basamos en la idea de un *cursor gráfico*. Este cursor contiene una pareja de coordenadas que describe su posición actual y un ángulo que describe su orientación. Necesitamos definir previamente la longitud inicial de A_0. Cada vez que "avancemos", calculamos la siguiente posición usando a) la posición actual del cursor como punto inicial, b) tomando un tercio de la longitud inicial, y c) aplicando trigonometría (senos y cosenos) con el ángulo del cursor. La razón de usar un tercio de la longitud es para reducir el tamaño de cada figura para que la imagen final se limite a una distancia adecuada. Si usáramos la misma longitud inicial para cada segmento, entonces obtendríamos una imagen bastante grande cuantas más iteraciones hiciéramos; es decir, cuanto mayor sea el valor de *n*. La imagen sería tan grande que posiblemente parte de ella sobresalga la pantalla. Por esta razón, optamos por dividir el segmento inicial en partes de menor longitud, para que la curva final sea más manejable.

Algoritmo

El algoritmo para trazar la curva de Koch se divide en dos partes. La primera parte, *Iniciar_Koch()*, es básicamente invocar la función recursiva, *Dibujar_Koch()*. La segunda parte, *Dibujar_Koch()*, es el algoritmo de la función recursiva en sí.

Para *Iniciar_Koch()*, el algoritmo es:

```
Iniciar_Koch( Punto_Inicial, Longitud, Número_Iteraciones )
1.  Declarar  Cursor  como un punto y un ángulo
2.  Inicializar  Cursor:
3.     Cursor.punto  <- Punto_Inicial
4.     Cursor.angulo <- 0
5.  Dibujar_Koch( Cursor, Longitud, Número_Iteraciones )
6.  Terminar
```

Como ángulo inicial, elegimos 0 radianes. Esto implica que estamos en dirección horizontal y con una orientación hacia la derecha.
Longitud es la distancia entre el punto inicial y final.

Para *Dibujar_Koch()*, el algoritmo es:

```
Dibujar_Koch( Cursor, Dist, N )
1.  Si  N = 0  entonces,                  // A(vanzar)
2.     Cursor.punto <- Avanzar( Cursor, Dist )
3.     Terminar
4.  Si no, entonces,                      // Recursividad: A->AIADDAIA
5.     Dibujar_Koch( Cursor, Dist/3, N-1 )              // A
6.     Cursor.angulo <- Cursor.angulo + π/3             // I
7.     Dibujar_Koch( Cursor, Dist/3, N-1 )              // A
8.     Cursor.angulo <- Cursor.angulo - π*2/3           // DD
9.     Dibujar_Koch( Cursor, Dist/3, N-1 )              // A
10.    Cursor.angulo <- Cursor.angulo + π/3             // I
```

```
11.    Dibujar_Koch( Cursor, Dist/3, N-1 )              // A
12.    Terminar
```

Dist es la distancia o longitud de cada segmento.
N es el número de iteraciones.

Para *Avanzar()*, el algoritmo es:

```
Real Avanzar( Cursor, D )
1.  Declarar  P  como un punto: (x,y)
2.  P.x <- Cursor.punto.x + D*cos( Cursor.angulo )
3.  P.y <- Cursor.punto.y + D*sen( Cursor.angulo )
4.  Dibujar_Linea( Cursor.punto.x, Cursor.punto.y, P.x, P.y )
5.  Terminar( P )
```

D es la distancia o longitud del segmento a trazar.
Dibujar_Linea() es la función básica, en cualquier API o librería gráfica, para trazar una línea recta desde un píxel a otro.

En el algoritmo, debemos actualizar el *Cursor* después de cada operación: avanzar o girar. Una vez acabada un avance, actualizamos la posición de *Cursor.punto* con la nueva posición (calculada). Para realizar un giro, agregamos o restamos el ángulo en *Cursor.angulo*.

Observaciones

Para trazar las líneas rectas, podemos usar directamente los valores de los píxeles. Sin embargo, como los píxeles deben ser valores enteros, no podemos guardarlos como tales en *Cursor.punto*. Si lo hiciéramos, entonces perderíamos información al truncarse la parte decimal. Esto implicaría que la imagen contendría errores visibles, especialmente al repetir muchas veces: $n > 1$. Para evitar este efecto, guardamos los píxeles en *Cursor.punto* como valores decimales. En el momento de trazar la línea recta, debemos convertir tales valores decimales a enteros usando cualquier método de redondeo. De esta forma, sólo truncamos los valores en el momento propicio (al trazar una línea recta), pero manteniendo la información en nuestro cursor gráfico y así no producir errores de aproximación.

En el algoritmo anterior, no hemos tenido en cuenta la orientación del eje-Y de la pantalla (en píxeles). Esto es porque la función *Dibujar_Linea()* se hace cargo de ello. De todas formas, podemos implementar este cambio de orientación con simplemente cambiar el signo del ángulo: girar a la izquierda es negativo y girar a la derecha es positivo. Según las reglas de trigonometría,
cos(-x) = cos(x), y
sen(-x) = -sen(x)

En nuestro algoritmo, esto implica que el valor de la coordenada X permanece invariado, mientras que el de la coordenada Y es de sentido negativo. Esto es justo lo que necesitamos, ya que la mayoría de los sistemas gráficos definen la orientación del eje-Y como positivo desde arriba a abajo, en vez de la orientación convencional del plano cartesiano.

Explicación Detallada

Para aquellos lectores que les interese conocer más acerca de las matemáticas relacionadas con estos conceptos, expondremos una explicación acerca de este fractal. No es necesario seguir esta explicación para dibujar fractales,

por lo que el lector puede saltarse este apartado.

La dimensión fraccional - también denominada *dimensión de capacidad* o *dimensión de Hausdorff* - para la curva de Koch se basa en la longitud y número de tramos de cada iteración. Dejemos que $\mathbf{N_n}$ sea el número de tramos, según la iteración **n**, y $\mathbf{L_n}$, la longitud de cada tramo, según la iteración **n**.

$N_0 = 1,$ (Imagen de la Figura 3)
$N_1 = 4,$ (Imagen de la Figura 4)
$N_2 = 16,$ (Imagen de la Figura 5)
$N_3 = 64,$
.
.
.
$N_n = 4^n,$
$L_0 = 1,$ (Imagen de la Figura 3)
$L_1 = 1/3,$ (Imagen de la Figura 4)
$L_2 = 1/9,$ (Imagen de la Figura 5)
$L_3 = 1/27,$
.
.
.
$L_n = 1/3^n = 3^{-n}$

El cálculo de la dimensión de la capacidad, $\mathbf{d_{cap}}$, es:

$$d_{cap} = -\lim_{n\to INF} \frac{\ln N_n}{\ln L_n} = -\lim_{n\to INF} \frac{\ln 4^n}{\ln 3^{-n}} = -\lim_{n\to INF} \frac{n * \ln 4}{-n * \ln 3}$$

$$= \lim_{n\to INF} \frac{\ln 4}{\ln 3} = \frac{\ln 4}{\ln 3} = 1{,}261859507...$$

(Nota: INF se refiere al concepto matemático de "infinito").

La dimensión de la curva de Koch es 1,262, aproximadamente. Podemos comprobar este valor aplicando la siguiente fórmula:

$N_n = L_n^{-d_{cap}} \Rightarrow N_n = (3^{-n})^{-1,262} \Rightarrow N_n = 4^n$

Aquí vemos que $\mathbf{d_{cap}}$ es el exponente para $\mathbf{L_n}$ que equivale a $\mathbf{N_n}$.

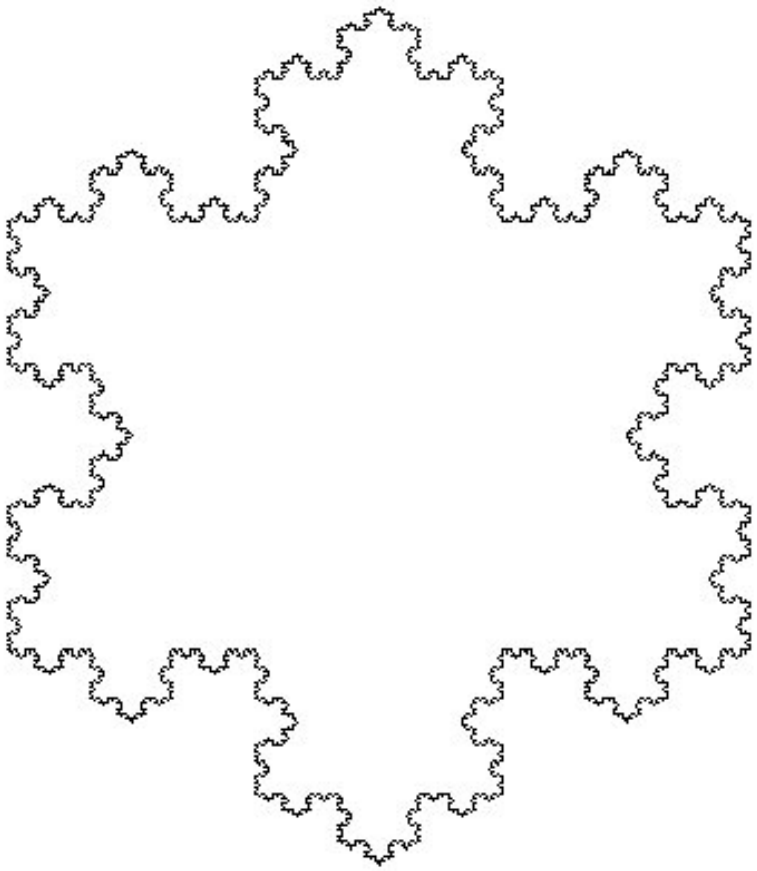
Ejercicios

Enlace al paquete de este capítulo.

1. Escribir un programa que dibuje la curva de Koch. Se puede implementar el algoritmo mostrado en el capítulo. Probad con varios valores de N; N=2,3,4. Después de muy pocas pasadas no se notará visualmente gran diferencia de detalle en la imagen. Probad con una resolución aceptable: 300x300 ó 500x500. También se puede cambiar el ángulo inicial de 0 radianes a $\pi/4$ radianes (=45°), o al que guste.

2. Uno de los ejemplos más populares es crear el Copo de Nieve de Koch. Esto se hace creando la curva de Koch basada en un triángulo equilátero en vez de una línea recta horizontal. Dicho de otra forma,
B -> $IA_nDDA_nDDA_n$, con un ángulo de 60° (=$\pi/3$ radianes) para formar el triángulo equilátero a partir de la "base".
A_0 -> Avanzar,
A_{n+1} -> $A_nIA_nDDA_nIA_n$,

Hemos agregado otra "regla", B, para describir la estructura que formará la base: un triángulo equilátero. Básicamente, estaremos dibujando 3 curvas de Koch situadas en cada lado del triángulo, con las "puntas" hacia fuera. Después de unas cuantas iteraciones, la figura dará forma a un copo de nieve.

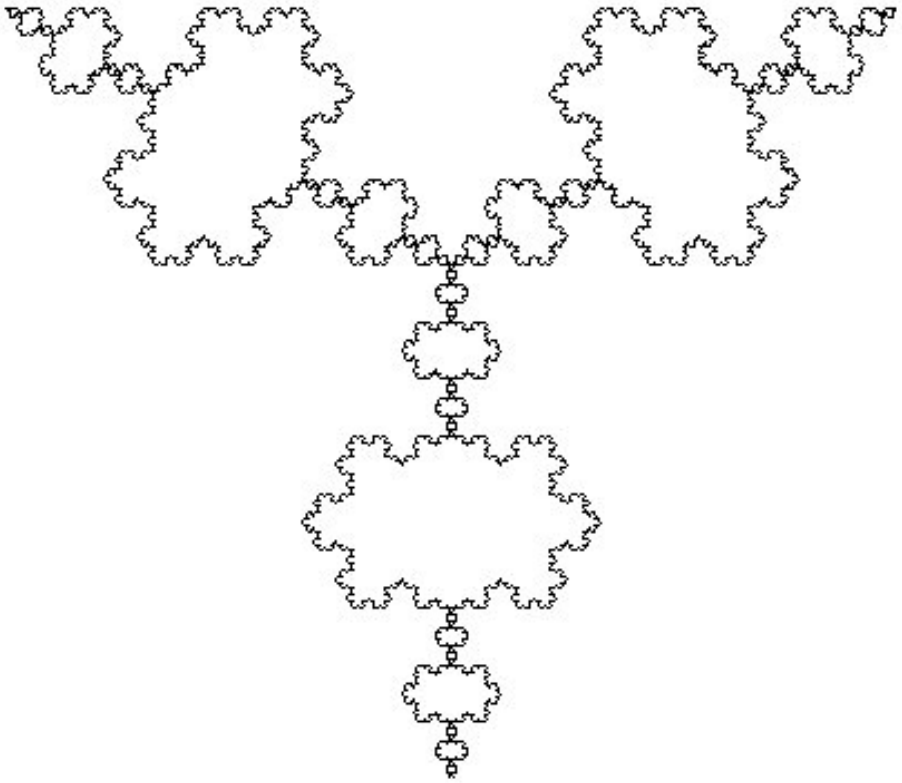


Copo de Nieve - N=5

El ejercicio consiste en crear un programa que dibuje el copo de nieve de Koch descrito anteriormente.

3. Otro ejemplo popular es realizar el Anti-Copo de Nieve de Koch. Esto consiste en dibujar las curvas de Koch con las puntas hacia el interior del triángulo al igual que tener un triángulo invertido; el triángulo se apoya con su pico y con un lado hacia en la parte superior en vez de en la inferior. Existen varias formas de realizar este fractal. Podemos optar por cambiar la regla de:
a) A_{n+1} para que la punta de la curva de Koch se oriente hacia la derecha. Es decir,

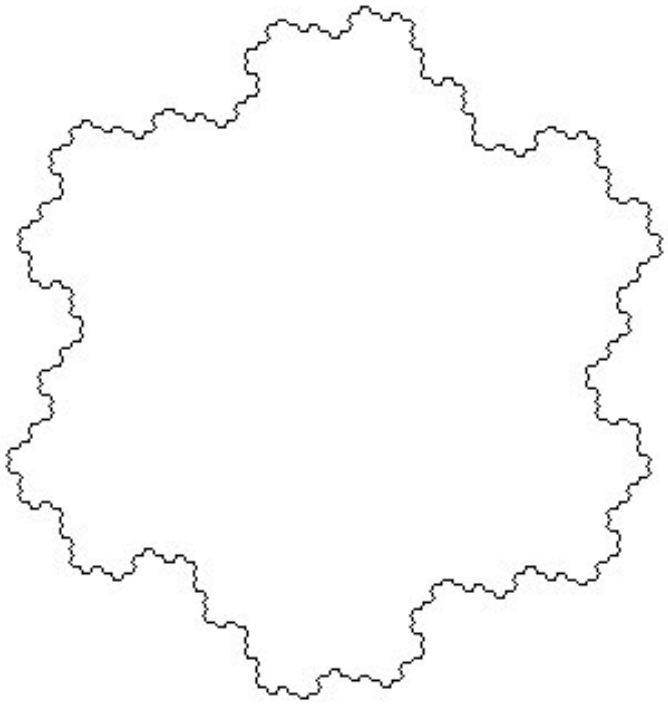
A_{n+1} -> $A_nDA_nIIA_nDA_n$, o
b) B para que la forma de dibujar el triángulo equilátero básico ya tenga una orientación inversa. Esto implicaría que,
B -> $IA_nIIA_nIIA_n$, a partir del "pico" del triángulo invertido.



Anti-Copo de Nieve - N=5

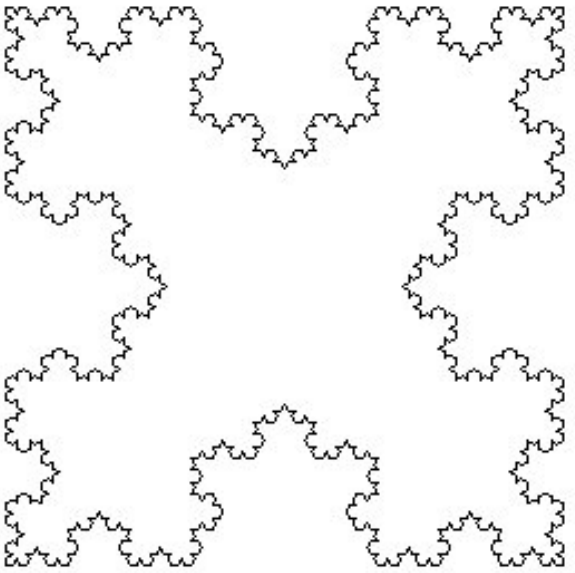
Escribir un programa que dibuje el anti-copo de nieve descrito en este ejercicio.

4. Con estas reglas de producción, podemos construir muchos tipos de figuras. Escribe un programa para dibujar cada figura descrita por las siguientes reglas de producción:
a) La Isla de Gosper,
B -> $A_nDA_nDA_nDA_nDA_nDA_n$, describe un hexágono regular,
A_0 -> Avanzar,
A_{n+1} -> $A_nIA_nDA_n$,
Todos los giros se hacen con un ángulo de 60° (= π/3 radianes).
Resolución: 400x400.
Longitud original (del Avance): 250 píxeles.
División del tramo: d=1/3.
Calcular A_4.



Isla de Gosper - N = 4

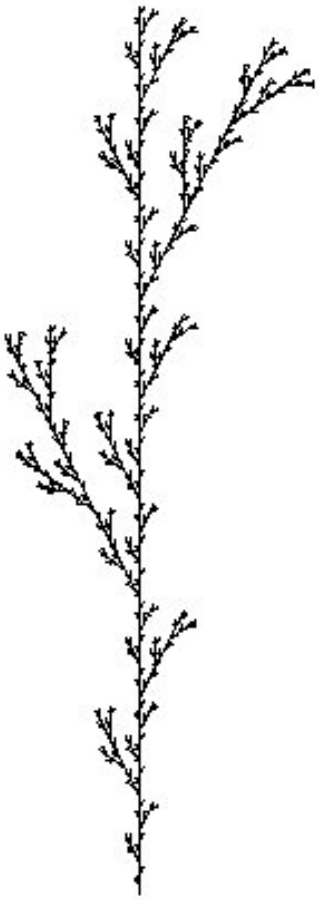
b) Fractal de Cesàro,
B -> $A_nIA_nIA_nIA_n$, con un ángulo de 90° (=π/2 radianes), describe un cuadrado,
A_0 -> Avanzar,
A_{n+1} -> $A_nIA_nDDA_nIA_n$,
Los giros de A_n se hacen con un ángulo de 60° (= π/3 radianes). Esta regla es la misma que la curva de Koch, pero la regla de la base es un cuadrado. Los picos de la curva de Koch apuntan al interior del cuadrado. Esto "restará" al cuadrado original. Obtendremos una imagen parecida a un copo de nieve.
Resolución: 400x400.
Longitud original (del Avance): 250 píxeles.
División del tramo: d=1/3.
Calcular A_4.



Fractal de Cesàro - N=4

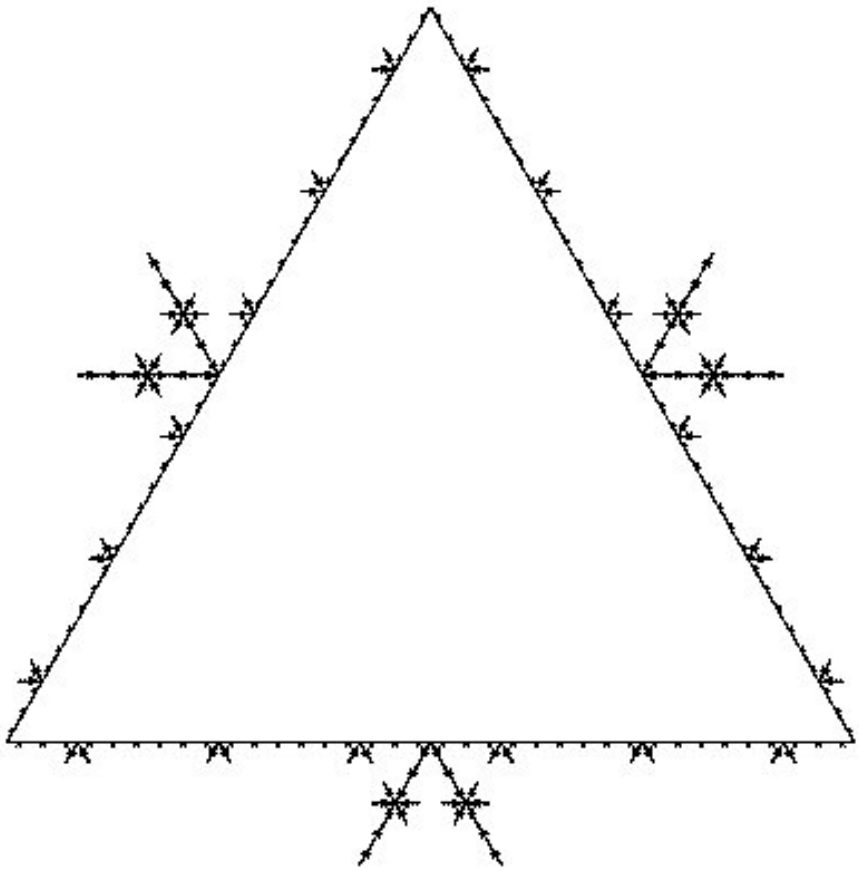
5. Una limitación, que podemos observar con las reglas de producción presentadas en este capítulo, es que son lineales. Seguimos avanzando, a medida que leemos - e interpretamos - los símbolos, hasta terminar según la profundidad en que nos encontremos. Ahora agregaremos la característica de *recordar* un lugar en nuestra regla y poder volver a ello. Podemos agregar otros dos símbolos a nuestra *gramática* que forman las reglas de producción; éstos son: **[** y **]**. El corchete abierto, **[**, representa que el *Cursor* es agregado a una pila. El corchete cerrado, **]**, indica que se saca - y se usa - el *Cursor* de la pila. Observad que la información guardada es tanto la posición como el ángulo del *Cursor*.

Escribir un programa que dibuje una figura para cada una de las siguientes reglas de producción:
a) Árbol sin hojas pero con muchas ramas,
A_0 -> Avanzar,
A_{n+1} -> $A_n[IA_n]A_n[DA_n]A_n$,
Ángulo inicial: 90° (=π/2 radianes) - para que el árbol esté "de pie".
Todos los giros se hacen con un ángulo de 27° (= 0,471239 radianes).
Resolución: 400x400.
Longitud original (del Avance): 400 píxeles.
División del tramo: d=1/3.
Calcular A_6.



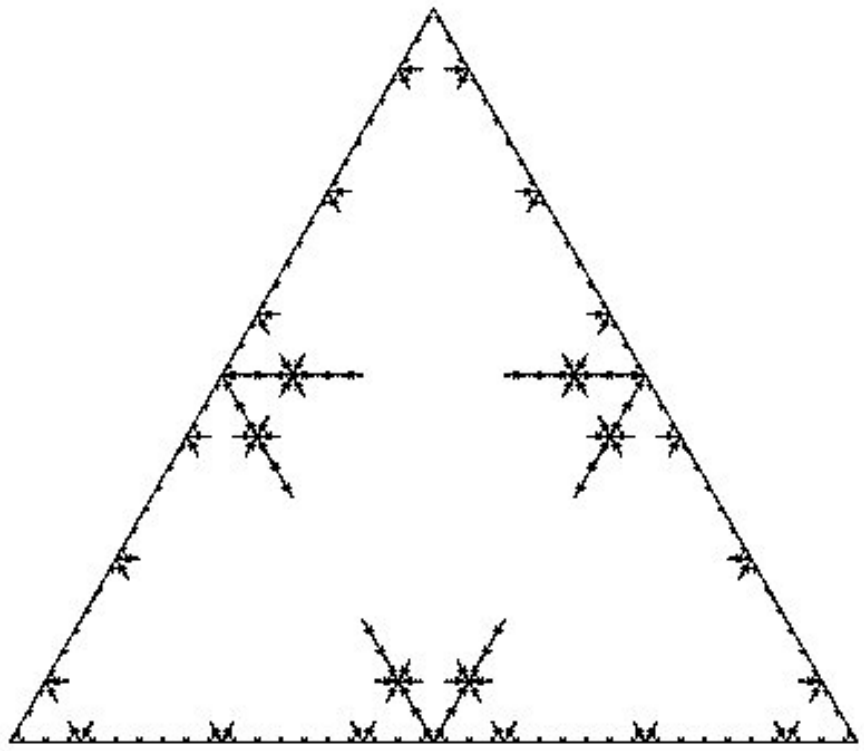
Árbol - N=6

b) Fractal de Hielo, basado en un triángulo,
B -> $IA_nDDA_nDDA_n$, con un ángulo de 60° para formar el triángulo equilátero,
A_0 -> Avanzar,
A_{n+1} -> $A_nA_nA_nIIA_nDDDA_nIIA_nDDDA_nIIA_nA_nA_n$,
Todos los giros se hacen con un ángulo de 60° (= $\pi/3$ radianes).
Resolución: 400x400.
Longitud original (del Avance): 380 píxeles.
División del tramo: d=1/6
Calcular A_4.



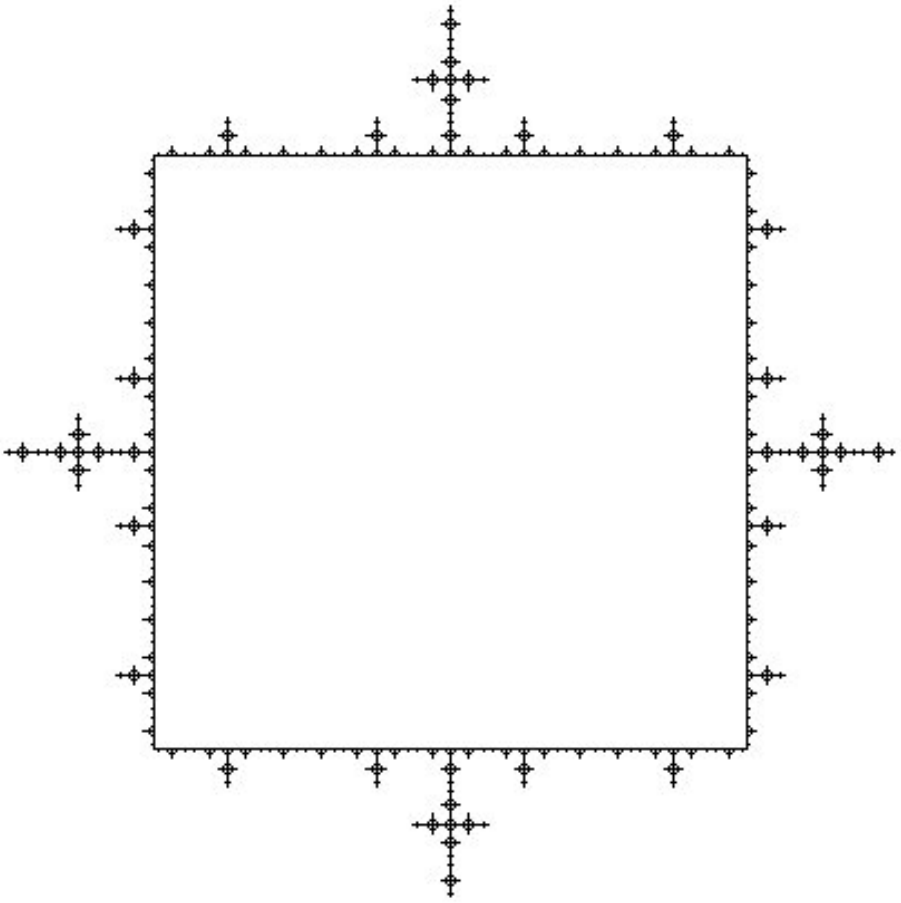
Fractal de Hielo - N=4

c) Fractal de Hielo, basado en un triángulo,
B -> $A_nIIA_nIIA_n$, con un ángulo de 60° para formar el triángulo equilátero,
Éste es el mismo fractal que el anterior en b), pero el fractal "crecerá" hacia el interior del triángulo.



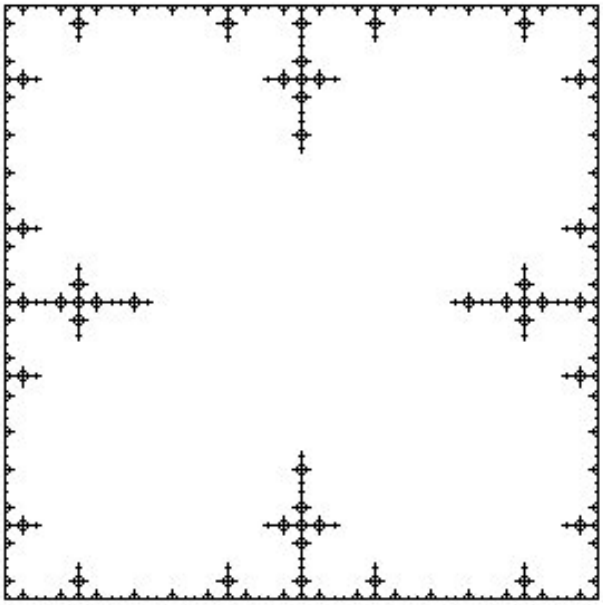
Fractal de Hielo - N=4

d) Fractal de Hielo, basado en un cuadrado,
B -> $A_nIA_nIA_nIA_n$, con un ángulo de 90° (=π/2 radianes), describe un cuadrado regular,
A_0 -> Avanzar,
A_{n+1} -> $A_nA_nIA_nDDA_nIA_nA_n$,
Todos los giros se hacen con un ángulo de 90° (= π/2 radianes).
Resolución: 400x400.
Longitud original (del Avance): 267 píxeles.
División del tramo: d=1/4.
Calcular A_4.



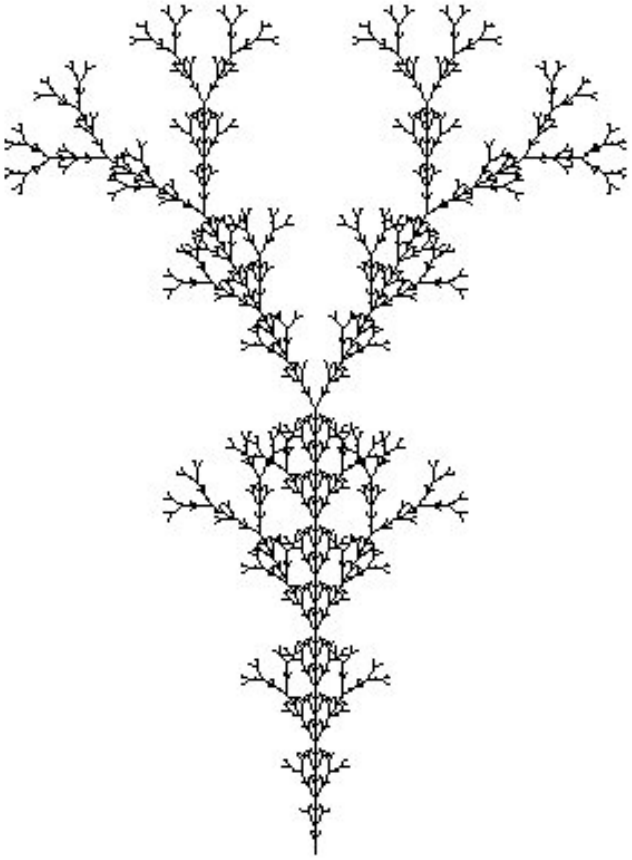
Fractal de Hielo - N=4

e) Fractal de Hielo, basado en un cuadrado,
B -> $A_nDA_nDA_nDA_n$, con un ángulo de 90° (=π/2 radianes), describe un cuadrado regular,
Éste es el mismo fractal pero el fractal "crecerá" en el interior del cuadrado.



Fractal de Hielo - N=4

f) Árbol con ramas y hojas,
A_0 -> Avanzar,
A_{n+1} -> $A_nA_n[IA_n][DA_n]$,
Ángulo inicial: 90° (=π/2 radianes) - para que el árbol esté "de pie".
Todos los giros se hacen con un ángulo de 30° (= π/6 radianes).
Resolución: 400x400.
Longitud original (del Avance): 200 píxeles.
División del tramo: d=1/2.
Calcular A_7.



Árbol - N=7

6. Ahora agregaremos otro símbolo a nuestra gramática. Se trata de la letra **E** que indica avanzar en el tramo que se encuentra pero en forma de espejo. Tenemos que invertir las instrucciones de la regla al igual que cambiar todos los casos de **I**zquierda a **D**erecha y de **D**erecha a **I**zquierda. Por ejemplo, si tenemos la siguiente regla:
A_0 -> Avanzar,

E_0 -> Avanzar,

A_{n+1} -> $IE_nA_nDA_nDE_nIA_n$, con un ángulo de giro de 90° (= π/2 radianes), y
E_{n+1} -> $A_nDE_nIA_nIA_nE_nD$

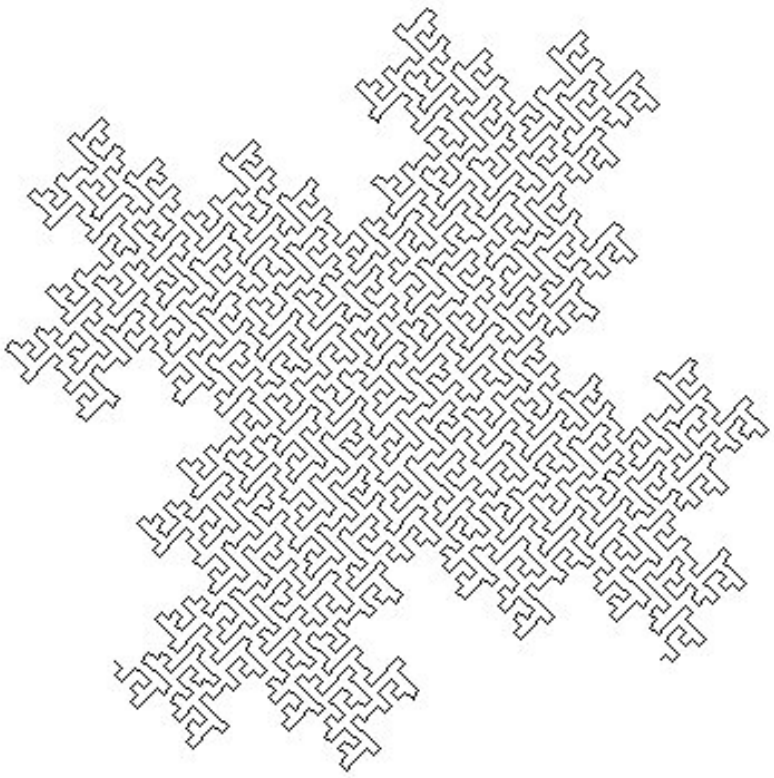
Como podéis observar, $\mathbf{E_{n+1}}$ invierte el orden de la regla de $\mathbf{A_{n+1}}$ y luego cambia todos los giros **I** y **D** a **D** e **I**, respectivamente.

Para A_1, la regla es simplemente: $IE_0A_0DA_0DE_0IA_0$, como E_0 es igual a A_0, entonces la regla es equivalente a: $IA_0A_0DA_0DA_0IA_0$
Para A_2, la regla se convierte en: $IE_1A_1DA_1DE_1IA_1$
Esta regla se expandirá a: I $A_0DE_0IA_0IA_0E_0D$ $IE_0A_0DA_0DE_0IA_0$ D $A_0DE_0IA_0IA_0E_0D$ D $IE_0A_0DA_0DE_0IA_0$ I $A_0DE_0IA_0IA_0E_0D$

Crear un programa para que trace cada uno de los siguientes fractales:
a) Curva basada en Peano,
A_0 -> Avanzar, y

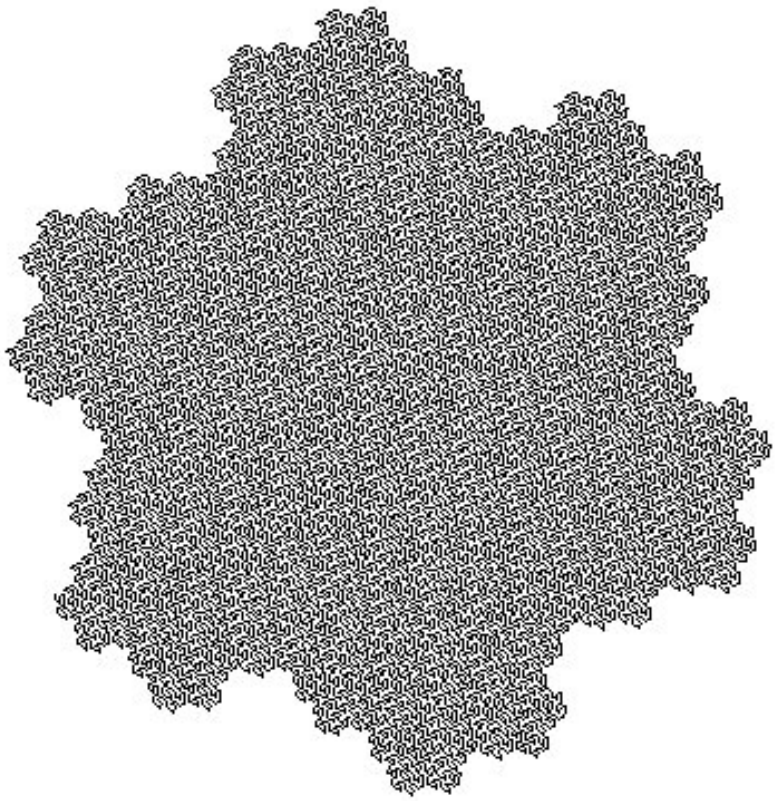
A_{n+1} -> $IE_nA_nDA_nDE_nIA_n$, con un ángulo de giro de 90° (= π/2 radianes)
Ángulo inicial: -132,825° (= -2,3182 radianes) o si lo preferís: 227,175° (= 3,9650 radianes).
Resolución: 400x400.

Longitud original (del Avance): 231 píxeles
División del tramo: d=√(1/5) = 0,44721.
Calcular A_5.



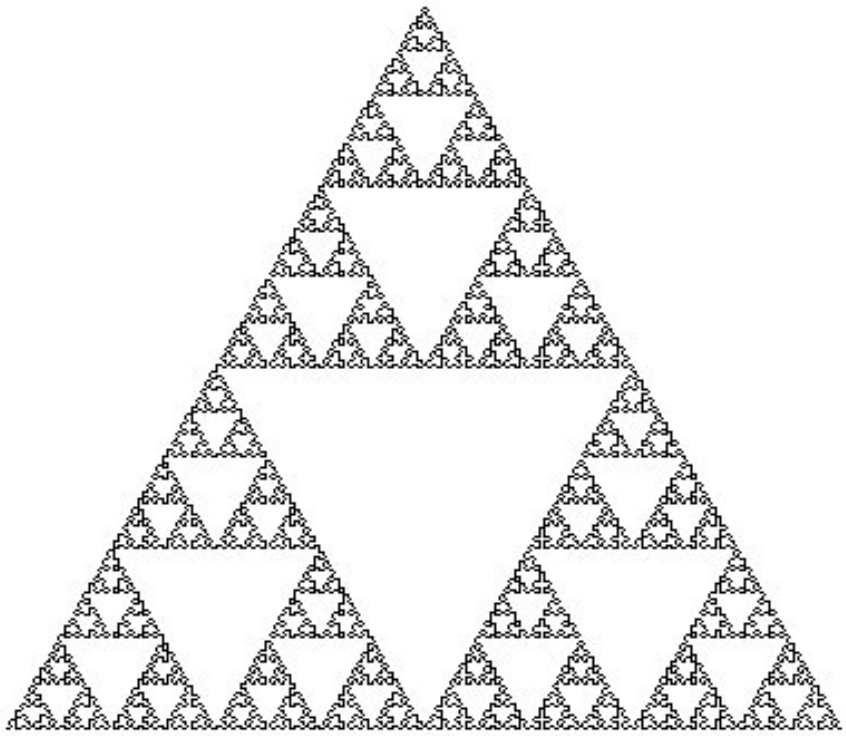
Curva basada en Peano - N=5

b) Curva de Peano-Gosper,
A_0 -> Avanzar,
A_{n+1} -> $A_nIE_nIIE_nDA_nDDA_nA_nDE_nI$,
Todos los giros se hacen con un ángulo de 60° (= π/3 radianes).
Resolución: 400x400.
Longitud original (del Avance): 300 píxeles.
División del tramo: d=√(1/7) = 0,37796.
Calcular A_5.



Curva de Peano-Gosper - N=5

c) Curva de Peano-Sierpinski,
A_0 -> Avanzar,
A_{n+1} -> $IE_nDA_nDE_nI$,
Todos los giros se hacen con un ángulo de 60° (= $\pi/3$ radianes).
Resolución: 400x400.
Longitud original (del Avance): 400 píxeles.
División del tramo: d=1/2.
Calcular A_8.



Curva de Peano-Sierpinski - N=8

Capítulo 3 - Fractales: Triángulo de Sierpinski

El segundo ejemplo de fractales que veremos es el triángulo de Sierpinski. Este triángulo también se basa en un método recursivo, como el ejemplo anterior. Comenzamos con un triángulo equilátero, como muestra la imagen de la *Figura 1*.

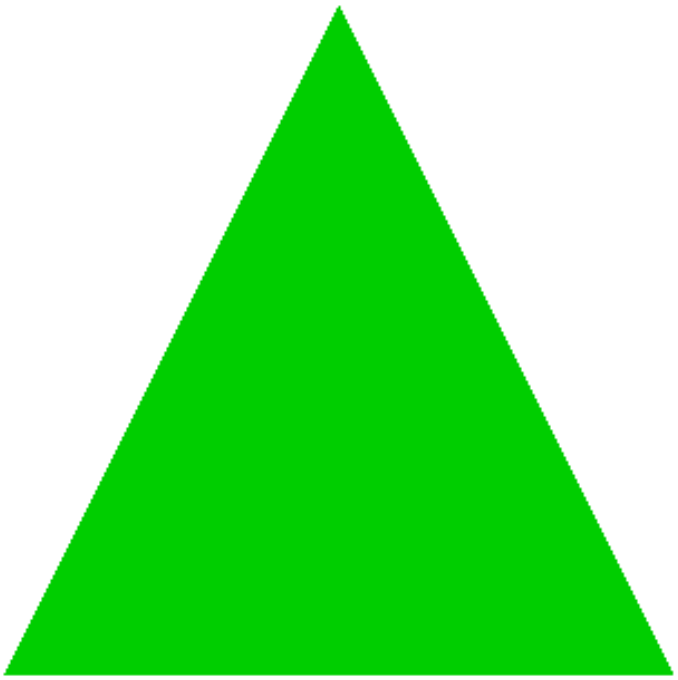


Figura 1

En cada pasada, dividimos el triángulo en tres triángulos más pequeños. Los lados de cada "sub-triángulo" equilátero tienen la mitad de longitud que los del triángulo original, el cual estamos dividiendo. Con la mitad de longitud, el área de cada subtriángulo es un cuarto del original. Por lo tanto, podemos rellenar el triángulo original con cuatro triángulos pequeños. En el fractal de Sierpinski, sólo nos interesamos con tres subtriángulos, por lo que el triángulo invertido en el centro es realmente un "agujero". Esto se observa más claramente en la *Figura 2*.

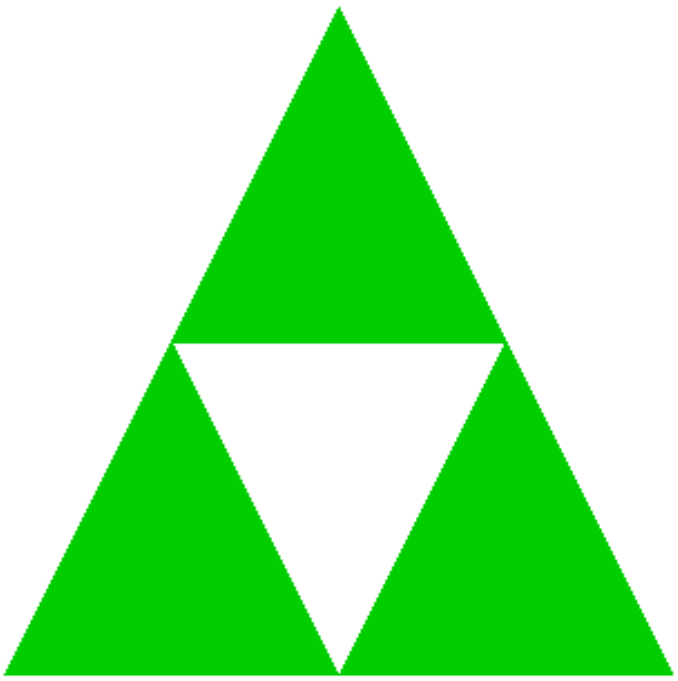


Figura 2

Con la tercera pasada, dividimos cada subtriángulo como si fuera el original. De nuevo obtenemos tres triángulos más pequeños y un agujero en el centro. El total será de nueve triángulos y cuatro agujeros. Esto se percibe más claramente en la *Figura 3*.

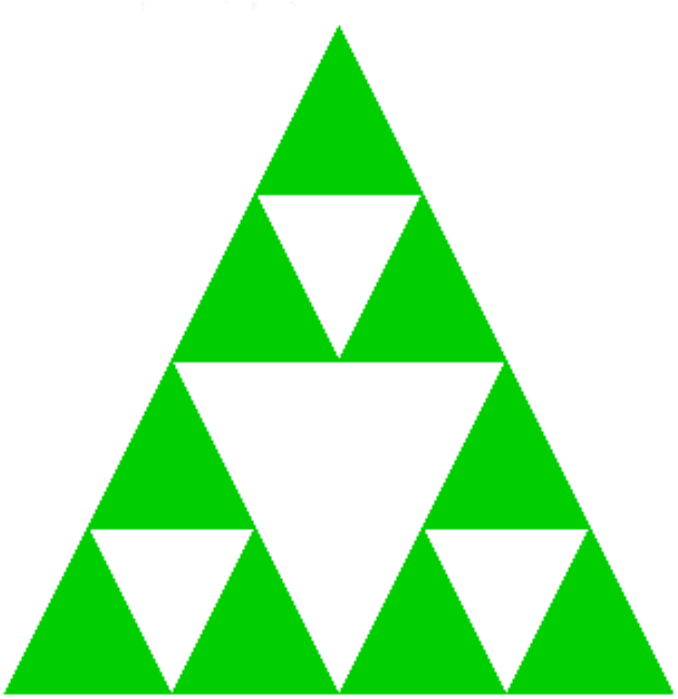


Figura 3

Nuevamente, podemos observar el elemento recursivo en este fractal, para generar el triángulo de Sierpinski. Comenzamos con tres puntos (A, B, y C) formando las coordenadas de los vértices del triángulo original. Luego, la función recursiva averiguará la mitad de cada lado (AB, BC, y CA) usando los vértices. El orden para subdividir cada triángulo es irrelevante: el izquierdo, el derecho, y luego el superior, o cualquier orden que se quiera seguir.

Algoritmo

El algoritmo para dibujar el triángulo de Sierpinski se puede separar en dos partes principales. La primera parte, *Iniciar_Sierpinski()*, es sencillamente la invocación a la función recursiva, *Dibujar_Sierpinski()*. La segunda parte es el algoritmo de la función recursiva en sí, *Dibujar_Sierpinski()*.

Para *Iniciar_Sierpinski()*, el algoritmo es:

```
1.  Declarar  A, B, y C  como puntos
2.  Inicializar  A, B, y C
3.  Dibujar_Sierpinski( A, B, C, Numero_Iteraciones )
4.  Terminar
```

A, *B*, y *C* son los vértices del triángulo original.

Para *Dibujar_Sierpinski()*, el algoritmo es:

```
Dibujar_Sierpinski( A, B, C, N )
```

```
1.  Si N = 0  entonces,          // Triángulo Mínimo
2.     Dibujar_Triangulo_Relleno( A, B, C )
3.     Terminar
4.  Si no, entonces,                    // Dividir en 3 triángulos
5.     AB <- Mitad( A, B )
6.     BC <- Mitad( B, C )
7.     CA <- Mitad( C, A )
8.     Dibujar_Sierpinski( A, AB, CA, N-1 )
9.     Dibujar_Sierpinski( AB, B, BC, N-1 )
10.    Dibujar_Sierpinski( CA, BC, C, N-1 )
11.    Terminar
```

A, *B*, y *C* son los vértices del triángulo o subtriángulo.
N es el número de iteraciones.

Para *Mitad()*, el algoritmo es:

```
Real Mitad( P1, P2 )
1.  Resultado.x <- (P1.x + P2.x) / 2
2.  Resultado.y <- (P1.y + P2.y ) / 2
3.  Terminar( Resultado )
```

En el algoritmo, no dibujamos nada hasta llegar al triángulo más pequeño; o sea, N = 0. En los otros niveles de profundidad, N > 0, sólo dividimos cada lado en tres triángulos más pequeños y calculamos las dimensiones de cada subtriángulo.

La función *Dibujar_Triangulo_Relleno()* hace referencia a una función de la librería o API gráfica para dibujar un triángulo según las vértices dadas en orden. Esto es, se trazan las líneas del primer vértice al segundo (A->B), del segundo al tercero (B->C), y del tercero al primero (C->A). El triángulo es dibujado y rellenado con los colores previamente establecidos.

Observaciones

Como sucedió con la curva de Koch, podemos usar directamente los valores de los píxeles para describir los vértices del triángulo. Sin embargo, como los píxeles deben ser valores enteros, no podemos guardarlos como tales en *A*, *B*, y *C*. Si lo hiciéramos, entonces perderíamos información al ser truncada la parte decimal. Esto implicaría que la imagen contendría errores visibles: los lados de los triángulos no son tan rectos, especialmente al repetir muchas veces: $n > 1$. Para evitar este efecto, guardamos los píxeles en *A*, *B*, y *C* como valores decimales. En el momento de dibujar el triángulo, debemos convertir tales valores decimales a enteros usando cualquier método de redondeo. De esta forma, sólo truncamos los valores en el momento propicio (al dibujar un triángulo), pero manteniendo la información y así no producir errores de aproximación.

En el algoritmo anterior, no hemos tenido en cuenta la orientación del eje-Y de la pantalla (en píxeles), esto es porque hemos usado píxeles directamente. Por lo tanto, estamos calculando y dibujando en el mismo plano: la pantalla o píxeles. Esto implica que no necesitamos realizar ningún tipo de conversión o cambio de coordenadas.

Explicación Detallada

Para aquellos lectores que les interese conocer más acerca de las matemáticas relacionadas con estos conceptos,

expondremos una explicación acerca de este fractal. La dimensión de capacidad para el triángulo de Sierpinski se basa en la longitud y número de triángulos pintados de cada iteración.

Dejemos que $\mathbf{N_n}$ sea el número de triángulos, según la iteración **n**, y $\mathbf{L_n}$, la longitud de cada lado, según la iteración **n**.

$N_0 = 1,$ (Imagen de la Figura 1)
$N_1 = 3,$ (Imagen de la Figura 2)
$N_2 = 9,$ (Imagen de la Figura 3)
$N_3 = 27,$
.
.
.
$N_n = 3^n,$
$L_0 = 1,$ (Imagen de la Figura 1)
$L_1 = 1/2,$ (Imagen de la Figura 2)
$L_2 = 1/4,$ (Imagen de la Figura 3)
$L_3 = 1/8,$
.
.
.
$L_n = 1/2^n = 2^{-n}$

El cálculo de la dimensión de la capacidad, $\mathbf{d_{cap}}$, es:

$$d_{cap} = - \lim_{n\to INF} \frac{\ln N_n}{\ln L_n} = - \lim_{n\to INF} \frac{\ln 3^n}{\ln 2^{-n}} = - \lim_{n\to INF} \frac{n * \ln 3}{-n * \ln 2} = \lim_{n\to INF} \frac{\ln 3}{\ln 2} = \frac{\ln 3}{\ln 2} = 1{,}584962501...$$

(Nota: INF se refiere al concepto matemático de "infinito").

El triángulo de Sierpinski comienza como un objeto de dos dimensiones; o sea, una superficie. En cada iteración, se le "agregan" agujeros al triángulo. Llevado a infinito, obtenemos más agujeros que superficie, hasta quedarnos con un objeto de líneas rectas. Sin embargo, una línea recta es un objeto unidimensional. Es decir, hemos ido de un objeto 2D a un objeto 1D. La pregunta principal es ¿qué dimensión tiene este objeto? Según hemos visto, la dimensión debe quedar entre 1 y 2 dimensiones. Esta idea conlleva a la idea de una dimensión fraccional. En el caso del triángulo de Sierpinski, obtenemos una dimensión aproximada de 1,6, que efectivamente queda dentro de nuestro intervalo de 1 y 2 dimensiones.

Para dar un ejemplo, como explicación, imaginaos que tenemos un queso suizo con más agujeros que queso, hasta tal punto que tenemos un hilo de queso, pero todo agujero.

Ejercicios

Enlace al paquete de este capítulo.

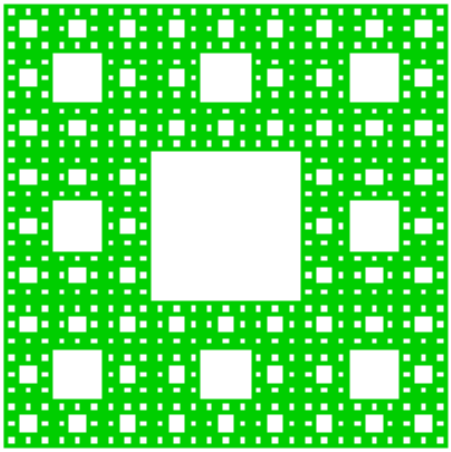
1. Escribir un programa que dibuje el triángulo de Sierpinski. Se puede usar el algoritmo presentado en este capítulo. Usar las siguientes restricciones y condiciones iniciales:
Resolución: 500x500.
Calcular, con N=5 iteraciones.

2. El triángulo de Sierpinski no tiene por qué ser equilátero. Construye el triángulo de Sierpinski para cada triángulo descrito por sus vértices, usa las siguiente restricciones:
Resolución: 500x500
Calcular, con N=4 iteraciones.

a) A = (0, 0), B = (499, 499), y C = (0, 499);
b) A = (250, 0), B = (375, 499), y C = (125, 499);
c) A = (300, 50), B = (400, 400), y C = (10, 200);

3. (Aleatorio) Escribir el triángulo de Sierpinski, pero ahora se perturbará los vértices. Cada vez que se calculen los puntos medios para cada subtriángulo, se agregará un valor aleatorio en una orientación aleatoria. La forma más sencilla de hacer esto es seleccionar un intervalo para la perturbación. Por ejemplo, elegimos un intervalo de 5 píxeles. Necesitamos generar números aleatorios entre -2 y +2. Si por casualidad generamos el valor 0 (cero), entonces no realizamos ninguna perturbación. Las perturbaciones se deberán hacer en ambas direcciones: eje-X y eje-Y. Para una coordenada (x, y) agregamos dos valores aleatorios *a* y *b* a cada elemento, obteniendo (x+a, y+b).

4. En lugar de usar triángulos, podemos usar cuadrados. Comenzamos con un cuadrado como figura original. En la primera iteración, dividimos nuestro cuadrado original en 9 cuadrados más pequeños de igual tamaño. El cuadrado central está vacío y queda como agujero, mientras que los demás son rellenados formando un borde. En las demás iteraciones, cada subcuadrado es sometido al mismo proceso. Este fractal se le denomina *Alfombra de Sierpinski*. La siguiente figura puede servir de ejemplo:



Escribir un programa para representar la alfombra de Sierpinski con las siguientes restricciones y valores iniciales:
Resolución: 500x500.
Calcular, con N=4 iteraciones.

5. Por supuesto, podemos seguir la misma idea básica, pero aplicada a otras figuras geométricas. Escribir un programa para generar fractales según las siguientes figuras, restricciones, y valores iniciales:

a) Figura: Pentágono regular.
Resolución: 500x500
Número de iteraciones: N=4.

b) Figura: Hexágono regular.
Resolución: 600x600
Número de iteraciones: N=4.

Capítulo 3 - Fractales: Mandelbrot

El tercer y último ejemplo de fractales que daremos es el famoso fractal de Mandelbrot. Este fractal se basa en una ecuación y un método iterativo. La imagen que mostramos **no** es una gráfica de una ecuación sino un conjunto de valores.

Partimos de la ecuación: $Z_{n+1} = Z_n^2 + C$, donde *Z* y *C* son números complejos y $n \geq 0$. Inicialmente, $Z_0 = C$. *Z* es la variable, *C* una constante, y *n* el índice para la secuencia y por tanto para la iteración. Brevemente, los números complejos se componen de dos partes distintas: la parte real ***x*** y la parte imaginaria ***y***. Podemos escribir un número complejo como z = x + iy, donde *x* e *y* son números reales e *i* es el número complejo definido como $i = \sqrt{-1}$. También se puede reescribir usando la siguiente notación: (x, y).

Para la ecuación de Mandelbrot, podemos reescribirla en la forma de sus componentes:

$$(x_{n+1}, y_{n+1}) = (x_n^2 - y_n^2 + x_c,\ 2x_n y_n + y_c)$$

Visto de otro modo:

$$x_{n+1} = x_n^2 - y_n^2 + x_c,$$
$$y_{n+1} = 2x_n y_n + y_c$$

donde $Z_n = (x_n, y_n)$ y $C = (x_c, y_c)$.

El método para obtener el fractal, basada en esta ecuación, trata de averiguar si un número complejo cualquiera para C pertenece al conjunto de Mandelbrot. Si C pertenece al fractal, entonces se colorea de negro (por ejemplo), y si no, pues de blanco. La forma de saber si C pertenece o no al fractal de Mandelbrot es calculando la siguiente iteración y comprobando su distancia a un valor limitante, *r*. Esto es,
$|Z_n| < r$, donde $|Z_n|$ es la distancia o longitud del número complejo.
Esto se puede reescribir como:
$|Z_n| = |\sqrt{(x_n^2 + y_n^2)}| < r$

Generalmente, se usa un radio (valor limitante) $r = 2$ para el fractal de Mandelbrot.

Ahora que tenemos las condiciones impuestas, deberemos iterar el proceso. Sin embargo, necesitamos otro valor limitante para saber cuándo terminar. Este valor deberá ser grande para dejar suficiente tiempo al proceso con el fin de comprobar que el valor de Z_n no se salga fuera del radio *r*. Podemos usar un nivel de tolerancia de 3.000, 10.000, o incluso 20.000. Pero la verdad es que este valor depende del área del fractal en que nos fijemos. Si magnificamos una parte del fractal es posible que perdamos algunos detalles porque nuestro nivel de tolerancia no era muy grande. La causa de esto es que algunos valores de Z_n sean malinterpretados como valores no pertenecientes al conjunto de Mandelbrot, cuando en verdad sí pertenecen.

Cambio de Coordenadas

Según vimos en el capítulo 2, debemos realizar un cambio de coordenadas, ya que los sistemas de coordenadas que usamos no son idénticos. Como nuestro proceso usa la fórmula de Mandelbrot, nos encontramos en el sistema de coordenadas complejas. Sin embargo, podemos pasarnos al sistema cartesiano con facilidad. El problema está en que al terminar nuestros cálculos, debemos representar *algo* en la pantalla. Por este motivo debemos cambiar del sistema cartesiano al sistema gráfico de la pantalla.

En el caso de generar el fractal de Mandelbrot, iremos píxel a píxel comprobando si éstos pertenecen al conjunto de Mandelbrot o no. Para hacer esto, debemos conocer previamente las dimensiones de nuestra vista. Como dijimos en el capítulo 2, no podemos representar todos los valores, porque éstos son infinitamente muchos, cuando sólo disponemos de la pantalla que es finito: limitado. Por lo tanto, debemos escoger las dimensiones para los valores de X y para los valores de Y. Luego, debemos convertir tales valores a sus equivalentes en píxeles.

Digamos que queremos representar la imagen entre x = [-6, +6] e y = [-4, +4] a una resolución de 500x500. Necesitamos saber qué representa cada píxel (x_p,y_p) en términos de estas dimensiones, ya que cada pareja de valores corresponderá a un valor de *C* en nuestra fórmula de Mandelbrot. Siguiendo nuestro ejemplo, obtenemos que,

$$dx_{up} = \frac{|x_{ui} - x_{uf}|}{anchura_p} = \frac{|-6 - 6|}{500} = \frac{12 \text{ unidades}}{500 \text{ píxeles}} = 0{,}024 \text{ unidades/píxel}$$

$$dy_{up} = \frac{|y_{ui} - y_{uf}|}{altura_p} = \frac{|-4 - 4|}{500} = \frac{8 \text{ unidades}}{500 \text{ píxeles}} = 0{,}016 \text{ unidades/píxel}$$

En nuestro ejemplo, cada píxel representa 0,024 unidades en el eje X y 0,016 unidades en el eje Y del plano cartesiano. Los valores de dx_{up} y dy_{up} son nuestros incrementos para comprobar cada píxel para *C* en nuestra fórmula de Mandelbrot.

Algoritmo

De nuevo, separaremos el algoritmo para dibujar el fractal de Mandelbrot en dos partes principales:

- La primera parte, *Iniciar_Mandelbrot()*, es el cálculo para cambiar de píxeles a unidades cartesianas y la invocación a la función iterativa, *Mandelbrot()*.
- La segunda parte es el algoritmo del proceso iterativo, *Mandelbrot()*.

Para *Iniciar_Mandelbrot()*, el algoritmo es:

1. Inicializar valores para: x_{ui}, x_{uf}, y_{ui}, y_{ui}, x_{pi}, x_{pf}, y_{pi}, y_{pf},

```
2.     num_max_iteraciones, radio, C
3.  anchura_u <- |x_uf - x_ui|
4.  altura_u  <- |y_uf - y_ui|
5.  anchura_p <- |x_pf - x_pi + 1|
6.  altura_p  <- |y_pf - y_pi + 1|
7.  dx_up <- anchura_u / anchura_p
8.  dy_up <- altura_u / altura_p
9.  Bucle: x_p <- x_pi hasta x_pf con incremento de 1
10.    Bucle: y_p <- y_pi hasta y_pf con incremento de 1
11.       C.x <- x_ui + x_p * dx_up
12.       C.y <- y_ui - y_p * dy_up
13.       Color <- Mandelbrot( C, num_max_iteraciones, radio )
14.       PonPixel( x_p, y_p, Color )
15. Terminar
```

- *C* es un tipo que puede almacenar una coordenada en unidades cartesianas - un número complejo.
- Necesitamos dos bucles anidadas porque necesitamos recorrer y comprobar todos los píxeles en pantalla.
- La función *PonPixel()* hace referencia a una función de la librería o API gráfica para establecer un color para un píxel determinado, según las coordenadas dadas.

Para *Mandelbrot()*, el algoritmo es:

```
Color Mandelbrot( C, max_iter, r )
1.  bEsMandelbrot <- Verdadero
2.  Z <- C
3.  Z_Sig <- Ecuacion( Z, C )
4.  Bucle: k <- 1 hasta max_iter  con incremento de 1 y
5.         mientras que  bEsMandelbrot = Verdadero
6.     Si Distancia( Z_Sig ) < r  entonces
7.        Z <- Z_Sig
8.        Z_Sig <- Ecuacion( Z, C )
9.     Si no, entonces
10.       bEsMandelbrot <- Falso
11. Si  bEsMandelbrot = Verdadero  entonces
12.    Terminar( Negro )
13. Si no, entonces
14.    Terminar( Blanco )
```

C, *Z*, y *Z_Sig* son tipos que pueden almacenar una coordenada en unidades cartesianas.

En este algoritmo, tenemos dos condiciones para terminar el bucle:
1. k > max_iter, y
2. bEsMandelbrot = Falso

Esta última condición se basa en que la distancia de Z ≥ r. Es decir, Z se dispara hacia el infinito.

Al final, la función *Mandelbrot()* terminará devolviendo o bien el color negro, si el valor de C pertenece al conjunto de Mandelbrot, o bien blanco, si el valor de C no pertenece a dicho conjunto.

Para *Ecuacion()*, el algoritmo es:

```
Complejo Ecuacion( Z, C )
1.  Resultado.x <- Z.x*Z.x - Z.y*Z.y + C.x
2.  Resultado.y <- 2*Z.x*Z.y + C.y
3.  Terminar( Resultado )
```

Aplicamos la fórmula $Z_{n+1} = Z_n + C$, pero descomponiéndola en partes reales e imaginarias para usarse en el plano cartesiano.

Para *Distancia()*, el algoritmo es:

```
Real Distancia( Z )
1.  Resultado <- Raíz_Cuadrada_de( Z.x*Z.x + Z.y*Z.y )
2.  Terminar( Resultado )
```

Observaciones

En este método, comprobamos cada píxel para saber si tal coordenada en el plano complejo-cartesiano corresponde al conjunto de Mandelbrot. Esto se realiza en base a su fórmula y según las condiciones limitantes. Este algoritmo utiliza métodos vistos en el trazado de una ecuación en el plano cartesiano. Sin embargo, en el trazado, sólo dibujábamos líneas que pertenecían a la línea. En el caso del fractal de Mandelbrot, debemos comprobar cada píxel.

En el algoritmo anterior, hemos tenido en cuenta la orientación del eje-Y de la pantalla (en píxeles). Esto lo hemos realizado con la siguiente fórmula:
$x_u = x_{ui} + x_p * dx_{up}$,
$y_u = y_{ui} - y_p * dy_{up}$

Fijémonos en el signo positivo para el valor de x_u, y el signo negativo para y_u. El signo positivo para x indica una orientación idéntica, mientras que el negativo de y indica una orientación contraria al sistema de coordenadas. Esto suele ser lo habitual en sistemas gráficos, pero si esto no es el caso, con simplemente cambiar los signos podemos elegir la orientación que nos sirva.

Otra observación es la complejidad del tiempo de ejecución del algoritmo. Con el algoritmo descrito anteriormente, observaremos una ralentización en su ejecución. Esto se debe a que el algoritmo sigue comprobando si cada valor permanece dentro de las restricciones dadas, hasta completar todas las iteraciones. Sin embargo, podríamos aplicar ciertos criterios según la "naturaleza" de la función base. Realizando comprobaciones de periocidad y dirección, podemos optimizar el proceso. El inconveniente se basa en que cada comprobación depende de la función en sí. Otro inconveniente está en que no todos los lectores saben manipular números complejos para implementar tales comprobaciones en sus programas.

Explicación Detallada

Fractales como los de Mandelbrot, Julia, y muchos más no son utilizables por la mayoría de las personas, que no tengan una buena base de matemáticas. Las imágenes basadas en tales fractales son astéticamente agradables. Otros usos se basan en la propiedad de autosemejanza. Hoy en día existen varios métodos de compresión de datos basados en fractales - esta cualidad de autosemejanza. Se puede alcanzar compresiones de hasta 5.000 a 1. Por supuesto, todo depende de la variedad de los datos al igual a poder encontrar la fórmula matemática para describir

tales conjuntos de datos.

Ejercicios

Enlace al paquete de este capítulo.

1. Escribir un programa que represente el fractal de Mandelbrot, con estas características:
x = [-1, 1], e y = [-1, 1]
Valor máximo de iteraciones: 3.000
Radio: 4
Resolución: 300 x 300

2. Podemos usar otras funciones para crear otros tipos de fractales. Realizar un programa que genere un fractal basado en cada una de las siguientes fórmulas, reemplazando el algoritmo *Ecuacion()* en el apartado **Algoritmo**.
Nota: Se aconseja usar alguna librería para manipular números complejos, especialmente su no se sabe hacer usando matemáticas. En un programa de C++, se puede usar la plantilla `complex<T>` declarada en `<complex>` (o `<complex.h>`). Se sugiere usar la clase `complex<double>` para crear cada variable compleja y los operadores sobrecargados para llevar a cabo las operaciones necesarias: `*`, `+`, `-`, `/`, `sin()`, `cos()`, `exp()`, etc..

a) $Z_{n+1} = conj(Z_n)^2 + C$
Nota: conj() se refiere al conjugado; esto es:
z = x + iy,
su conjugado es:
z' = x - iy

b) $Z_{n+1} = sen(Z_n / C)$

c) $Z_{n+1} = C*exp(Z_n)$

d) $Z_{n+1} = Z_n^3 + C$

e) $Z_{n+1} = C*Z_n^2$

f) $Z_{n+1} = C*senh(Z_n)$

g) $Z_{n+1} = C*sen(Z_n)$

h) $Z_{n+1} = C*exp(conj(Z_n)) * C*exp(Z_n) = C^2*exp(2x)$
Nota: x indica la parte real de Z_n

i) $Z_{n+1} = sen(Z_n) + Z_n^2 + C$

j) $Z_{n+1} = cosh(Z_n) + Z_n^2 + C$

k) $Z_{n+1} = exp(conj(Z_n)) + conj(Z_n)^2 + C$

l) $Z_{n+1} = ln(Z_n) + Z^2 + C$
Nota: **ln** hace alusión al logaritmo neperiano o natural

Observación: Podemos crear cualquier función, especialmente tomando como modelo:

$Z_{n+1} = f(z) * Z_n$, donde f(z) es cualquier función trigonométrica compleja;
por ejemplo,
f(z) = cos(z),
f(z) = tan(z),
f(z) = 2*sen(z)*cos(z),
etc.

Advertencia: Es posible que, en algunas librerías estándares, las implementaciones de algunas funciones no den buenos resultados. Por lo tanto, usar las siguientes aproximaciones a:

```
double _exp( double x )
{
  return 1.0 + x*(1.0 + x*(0.5 + x*(1.0/6.0 + x*(1.0/24.0 + x/120.0))));
}

double _sinh( double x )
{
  return x != 0.0 ? (_exp(x) - _exp(-x)) / 2.0 : 0.0;
}

double _cosh( double x )
{
  return x != 0.0 ? (_exp(x) + _exp(-x)) / 2.0 : 1.0;
}

double _sin( double x )
{
  double x_sqr = x*x;

  return x*(1.0 - x_sqr*(1.0/6.0 + x_sqr*(1.0/120.0 - x_sqr/5040.0)));
}

double _cos( double x )
{
  x *= x;

  return 1.0 - x*(0.5 + x*(1.0/24.0 - x/720.0));
}
```

3. Se aconseja crear un puntero a una función, para luego poder cambiar de función fácilmente, sin tener que reprogramar el algoritmo. Por ejemplo, el código parcial en C++, puede ser el siguiente:

```
typedef Punto (*f_z)( Punto, Punto );

Punto mandelbrot( Punto Z, Punto C );

void fractal( f_z f, Punto Z_0, Punto C, unsigned long num_iteraciones,
double r )
{
  ...
  for( k=2; k<=num_iteraciones && !bEsMandelbrot; k++ )
    if( (d=dist(Z_sig)) < r )
    {
      Z = Z_sig;
```

```
      Z_sig = f( Z, C );  /* Invocamos la función general */
    }
    else bEsMandelbrot = true;
  ...
}

void Dibujar(void)
{
  ...
  fractal( mandelbrot, Z_0, C, num_iteraciones, r );
}
```

Podemos ver que sólo tenemos que pasar el puntero de la función que queramos mostrar. El algoritmo implementado en `fractal()` aplicará cualquier función dada.

4. Las imágenes interesantes de los fractales se encuentran en la "costa" o borde del fractal. Aquí es donde se encuentran muchas bahías, ríos, afluentes, y deltas. Crear un programa para facilitar el engrandecimiento (o zoom) de cualquier área de un fractal. Esto se puede lograr, calculando las dimensiones originales de la imagen, un factor de engrandecimiento, y un punto central a tal área engrandecida.

Por ejemplo,
Si comenzamos con una dimensiones de:
x = [-1, +3] e y = [0, +2],

entonces podemos hacer un zoom al área como punto central (0,5, -0,5) y con un factor de zoom de 3,0. Es decir, nuestro área nueva será 3 veces menor que la original, con un punto central de (0,5, -0,5). Esto ser haría de la siguiente forma:
1. Mudamos una de las esquinas de nuestro área rectangular al origen: (0, 0). Esto se puede calcular fácilmente: Elegimos la esquina inferior izquierda: (-1, 0). Ahora desplazamos las dimensiones, para que esta esquina se sitúe en el origen, (0, 0):
x = [-1-(-1), +3-(-1)], e
y = [0-0, +2-0],

Obtenemos,
x = [0, +4], e
y = [0, +2],

2. Ahora cambiamos las dimensiones según el factor de zoom: 3,0. Como se trata de un zoom hacia dentro, estamos reduciendo el tamaño por 3,0, que es lo mismo que dividir las longitudes entre 3,0. Esto sería:
x = [0/3,0, 4/3,0], e
y = [0/3,0, 2/3,0],

Resultando en,
x = [0,0, 1,3333], e
y = [0,0, 0,6666],

3. Ya que las dimensiones han sido reducidas, ahora podemos desplazar las dimensiones de nuestro área según el punto central dado: (0,5, 0,5). Esto no es más que una suma:
x = [0,0+0,5, 1,3333+0,5], e
y = [0,0-0,5, 0,6666-0,5],

obtenemos que,
x = [+0,5, +1,8333], e
y = [-0,5, +0,1666],

Recalculando el fractal con estas nuevas dimensiones, obtendremos una imagen de un área reducida 3 veces y centrada en el punto (0,5, -0,5).

5. Si el lector tiene experiencia en programar aplicaciones interactivas, entonces podemos crear un programa que permita al usuario elegir el área a investigar mediante la creación de un rectángulo pequeño en la imagen. Por ejemplo, al pinchar el botón izquierdo del ratón en la imagen, podría comenzar a crear un rectángulo arrastrándolo. Dicho rectángulo puede resultar al soltar tal botón izquierdo.

Este rectángulo ya contiene las nuevas dimensiones de la imagen, para realizar el zoom. Lo único que tendríamos que hacer es convertir los valores de los píxeles a su representación en coordenadas (matemáticas) del plano complejo.

6. Aún no hemos visto la forma de manipular colores; esto se trata en el siguiente capítulo 4. De todas formas, es una parte importante de la estética de la imagen. Un método popular se basa en dar un color a un píxel el cual representa un número complejo (coordenada) que **no** pertenece al conjunto del fractal. El criterio de dar un color u otro se basa en la distancia de tal coordenada desde su posición anterior (dentro del conjunto del fractal) a su posición actual (fuera del conjunto del fractal). Restringiendo tal distancia a un valor entre el intervalo: [0,00, 1,00], podemos usar un mapa de colores. Con este mapa podemos asignar o calcular un color, de acuerdo a nuestra gama de colores, según un valor entre el intervalo anterior. El tema de mapa de colores es tratado en el siguiente capítulo 4.

Podemos hacer el siguiente cálculo para obtener un valor, a modo de parámetro para nuestro mapa de colores, en el intervalo [0,00, 1,00]:
`mapa( r / d )`, donde `d` ≥ `r`
`d` es la distancia entre las dos últimas coordenadas, y
`r` es una constante que indica el radio del fractal.

Si el lector no tiene suficientes conocimientos para usar colores, entonces es recomendable pasar al siguiente capítulo 4, y luego volver a este ejercicio, cuando sepa la forma de manipular colores.

Capítulo 4 - Nubes

En este capítulo, veremos la forma de generar nubes blancas con un fondo de un cielo azul. Esto lo haremos manipulando los píxeles y los colores, según unas fórmulas sencillas. Por lo tanto, daremos una breve explicación acerca de los colores, antes de dar paso a la explicación y al algoritmo para crear nubes.

Colores

En el capítulo 1, se dio una breve explicación de los colores y los modelos de colores. En nuestra discusión de programación de gráficos, usaremos el modelo RGB. Intentamos modelar la luz descomponiéndola en tres colores primarios; también llamados estímulos tricolores. Sumando los tres estímulos, obtendremos un color o, mejor dicho, un tono de un color determinado. En sistemas gráficos, como ordenadores, cada estímulo o intensidad de un tricolor es representado por un valor numérico entero no negativo, que es limitado por una cantidad de bits. Dependiendo de la cantidad de bits, podremos usar más o menos colores a nuestra disposición. En este tema, al referirnos a varios miles de colores, realmente nos referimos a varios tonos de ciertos colores. Sin embargo, como el ordenador se basa en valores numéricos, cada número y, por tanto, cada combinación de números es diferente. Esto implica que los colores y tonos de colores representados por tales números son también diferentes colores, desde el punto de "vista" del ordenador o sistema gráfico.

Por ejemplo, con una cantidad de 15 bits para guardar la información de cada color, nos ofrece una gama de 2^{15} = 32.768 colores diferentes. De igual forma, una cantidad de 24 bits para crear un color, obtendremos una cantidad de 2^{24} = 16.777.216 colores diferentes. Esta cantidad de bits para cada color se denomina ***profundidad de colores***, del inglés *colour depth*. Como cada píxel puede tener un color diferente, se suele hablar de las prestaciones de un sistema gráfico al hablar de ***bits por píxel***, o simplemente **bpp**.

Al hablar de 15 bpp, estamos tratando un sistema gráfico con una gama total de 32.768 colores y por tanto, cada píxel contiene 15 bits para describir su color correspondiente. Si la imagen a mostrar es de dimensión, 500x500, entonces dicha imagen tiene un tamaño de 500x500x15 = 3.750.000 de bits que equivale a 468.750 bytes que son unos 458 KB aproximadamente. Ahora bien, en memoria, se suele usar bytes enteros. Por lo tanto, no podemos tratar 15 bits, pero sí 16 bits, ya que son 2 bytes enteros. Recalculando lo anterior con 16 bpp, obtenemos que, nuestra imagen es de unos 488 KB, aproximadamente. Esta profundidad de colores se suele llamar "color [de nivel] alto" del inglés "high colour"; productos estadounidenses lo escriben como "hicolor". Con una imagen de 500x500 á 24 bpp, obtenemos un tamaño de unos 732 KB. Una profundidad de 24 bpp se llama "color auténtico", del inglés "true colour", ya que se usa como base para mostrar imágenes fotorrealistas.

Modelo RGB

Vamos a tratar el modelo RGB con una profundidad de 24 bpp. Por lo tanto, cada intensidad - rojo, verde, y azul - es controlada por valores numéricos de 8 bits por cada intensidad. Visto de otra forma, las intensidades de rojo, verde, y azul tienen cada una un intervalo de 0 á 255 (ambos incluidos). Un valor de 0 indica la intensidad mínima; o sea, el color está "apagado". Un valor de 255 indica una intensidad máxima.

Combinando valores para cada intensidad, podemos recrear la mayoría de los colores visibles. Por ejemplo,
Rojo = 255, Verde = 0, Azul = 0,
Obtenemos el color rojo, ya que la intensidad del rojo es máxima, y las otras son mínimas (apagadas).

Otro ejemplo,
R=127, V=127, A=127,
Obtendremos un color gris, ya que todas las intensidades son iguales y a la mitad del intervalo. Por consiguiente, el color negro es:
R=0, V=0, A=0,
y el color blanco es:
R=255, V=255, A=255

Algunos sistemas y librerías gráficas, usan porcentajes en lugar de valores enteros. Esto es para no tener que definir los colores con acorde a la profundidad. Al usar valores numéricos, tendremos un problema si queremos cambiar la profundidad, o incluso si el sistema gráfico no alcanza la profundidad establecida por el programa o por la imagen. Algunos sistemas gráficos aplicarán una fórmula para averiguar un color aproximado al requerido. Si usamos un valor decimal a modo de porcentaje, entonces no tendremos muchos problemas de compatiblidad entre diferentes profundidades, pero consecuentemente se tendrá que convertir los porcentajes a valores numéricos siempre. Esto supone un gasto añadido de tiempo al mostrar la imagen.

El uso de porcentajes describe cada color indirectamente, mientras que el uso de valores enteros, dentro del intervalo de bits, describe cada color directamente según la profundidad establecida.

Abajo podemos practicar combinando cada una de las intensidades - Rojo, Verde, y Azul - para formar un color.

Applet para Colores

Nota: Se requiere la máquna virtual de Sun (Sun VM) para poder ejecutar este applet de Java.

Paletas

Vamos a hablar de paletas de colores, pero no entraremos en mucho detalle. Los pintores suelen usar unos cuantos colores básicos para dibujar sus cuadros. Tales colores se colocan en una paleta. Luego, se mezclan los colores básicos en la misma paleta para crear otros tonos y mezclas. El tema de gráficos generados por computación se sirve de este concepto de una paleta.

En algunos sistemas gráficos, especialmente antiguos, en lugar de manipular los colores directamente con el modelo RGB, se usaban números enteros a modo de índices. Previamente, se establecía todos los colores que se iban a usar para una imagen o programa. Básicamente, se crean los colores y se guardan en una lista. Al usar funciones gráficas, los píxeles contienen índices que indirectamente hacen referencia a esta lista o tabla donde se guardan los colores. Siguiendo el ejemplo en el apartado anterior: 500x500x16, digamos que tenemos una paleta de 8 bits. Esto supone que tenemos una paleta de 256 colores y por tanto 256 valores a modo de índices. Para cada píxel, sólo guardamos el índice del color y no el color en sí. Por lo tanto, 500x500x8 = 2.000.000 bits = 250.000 bytes, que es aproximadamente, 244 KB. Por supuesto, tenemos que agregar la paleta de 256 colores x 16 bpp = 4.096 bits = 512 bytes. Al final, la imagen ocupará aproximadamente unos 245 KB. Comparando este tamaño con el de la imagen usando colores directamente, 488 KB, vemos que podemos ahorrarnos una cantidad de espacio considerable.

Generar Nubes

Ahora podemos dar paso a la explicación para generar nubes. Lo que hacemos es elegir un color aleatoriamente para cada uno de los 4 puntos que están en las 4 esquinas de nuestra imagen rectangular:
(x_i, y_i), (x_i, y_f), (x_f, y_i), y (x_f, y_f)

La *figura 1* muestra los 4 puntos en las esquinas de nuestra imagen rectangular.

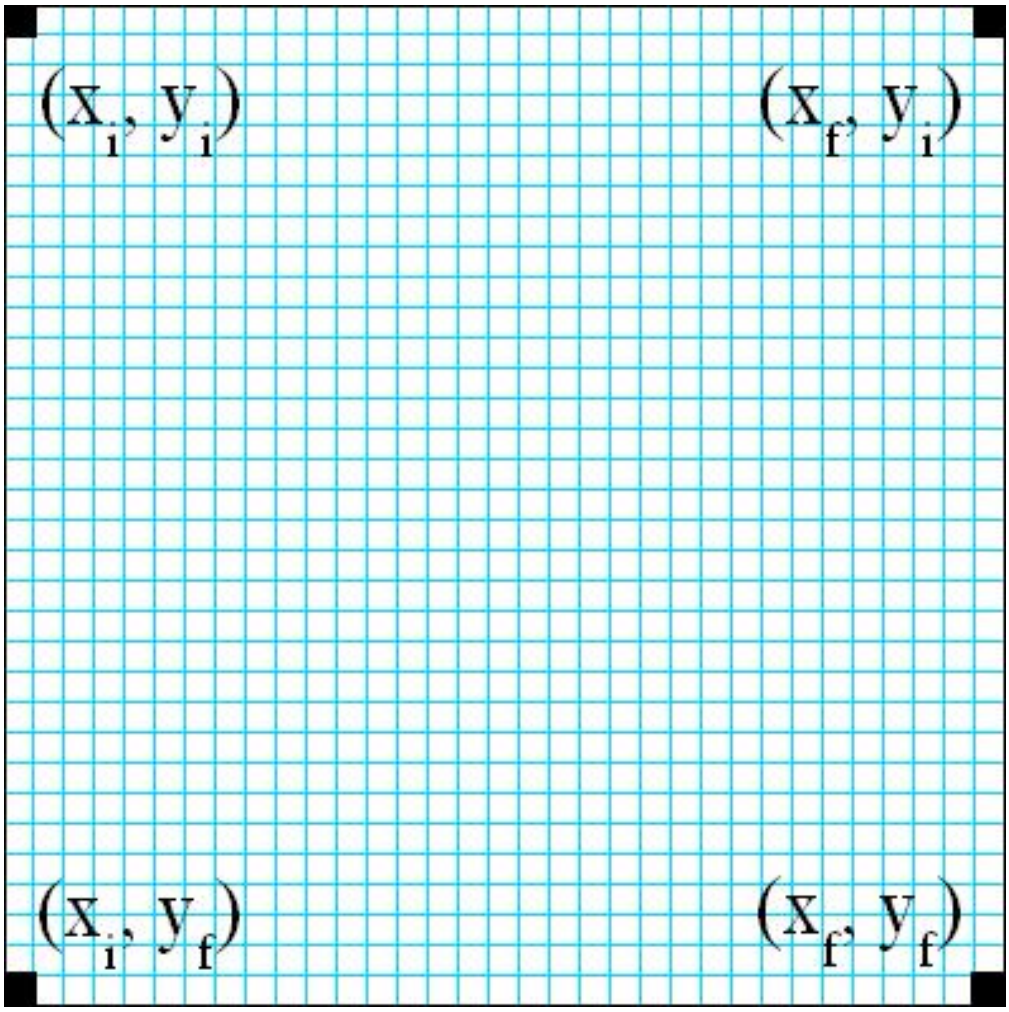


Figura 1 - Las 4 esquinas

Luego, calcularemos un 5º punto, que se situará en el centro del rectángulo: **(x',y')** = $((x_i+x_f)/2, (y_i+y_f)/2)$, el cual contendrá la media de los colores de los 4 puntos (o esquinas) obtenidos previamente. Con las coordenadas de este 5º punto junto con las coordenadas de los 4 puntos de las esquinas, podemos calcular otros 4 puntos medios para cada lado de nuestra imagen rectangular: (x', y_i), (x_f, y'), (x', y_f), y (x_i, y')

La *figura 2* muestra las nuevas coordenadas calculadas a partir de las 4 esquinas dadas:

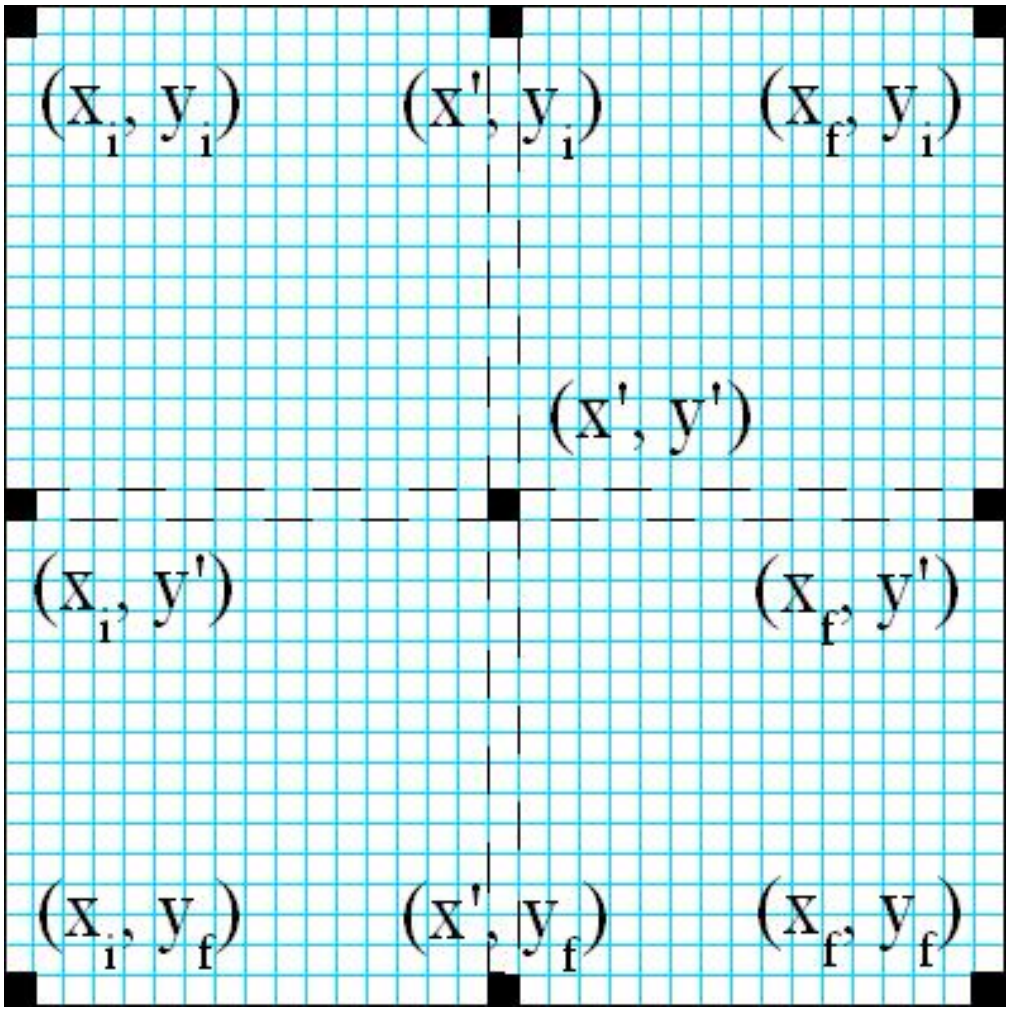


Figura 2 - 9 puntos totales

Para calcular el color de cada uno de estos puntos medios, calculamos la media de los colores de cada color de las 2 coordenadas de cada lado. Sin embargo, no obtendremos un resultado aleatorio, por lo que agregamos algo de aleatoriedad a la hora de calcular el nuevo color para el punto medio. El método de perturbación o aleatoriedad agregada se basa en la distancia entre las 2 coordenadas originales. Cuanto más distantes entre sí estén las coordenadas, mayor perturbación existirá. La idea es viceversa: cuanto menor sea la distancia entre las coordenadas, menor será el valor aleatorio. Esto resultará en agrupaciones de colores parecidos, cuanto menor sea la distancia, y por tanto, tendremos colores más dispares, cuanto más alejados estén las coordenadas entre sí. Para llevar a cabo este método de perturbación, el valor generado aleatoriamente será el parámetro para conseguir un color de nuestro mapa de colores; esto se explica más abajo.

Al tener 9 puntos: 4 esquinas, 4 puntos medios en cada lado, y 1 punto central, nuestra imagen rectangular puede divididirse en 4 rectángulos más pequeños. Recursivamente, repetimos el mismo proceso, aplicándolo a cada uno de los cuatro rectángulos o cuadrantes. Seguimos repitiendo tal proceso hasta que todos los píxeles de la imagen tengan asignados un color. Podemos ver la siguiente iteración en la *figura 3*, subdividiendo nuestra imagen rectangular en 4 áreas rectangulares para aplicar el mismo método a cada una:

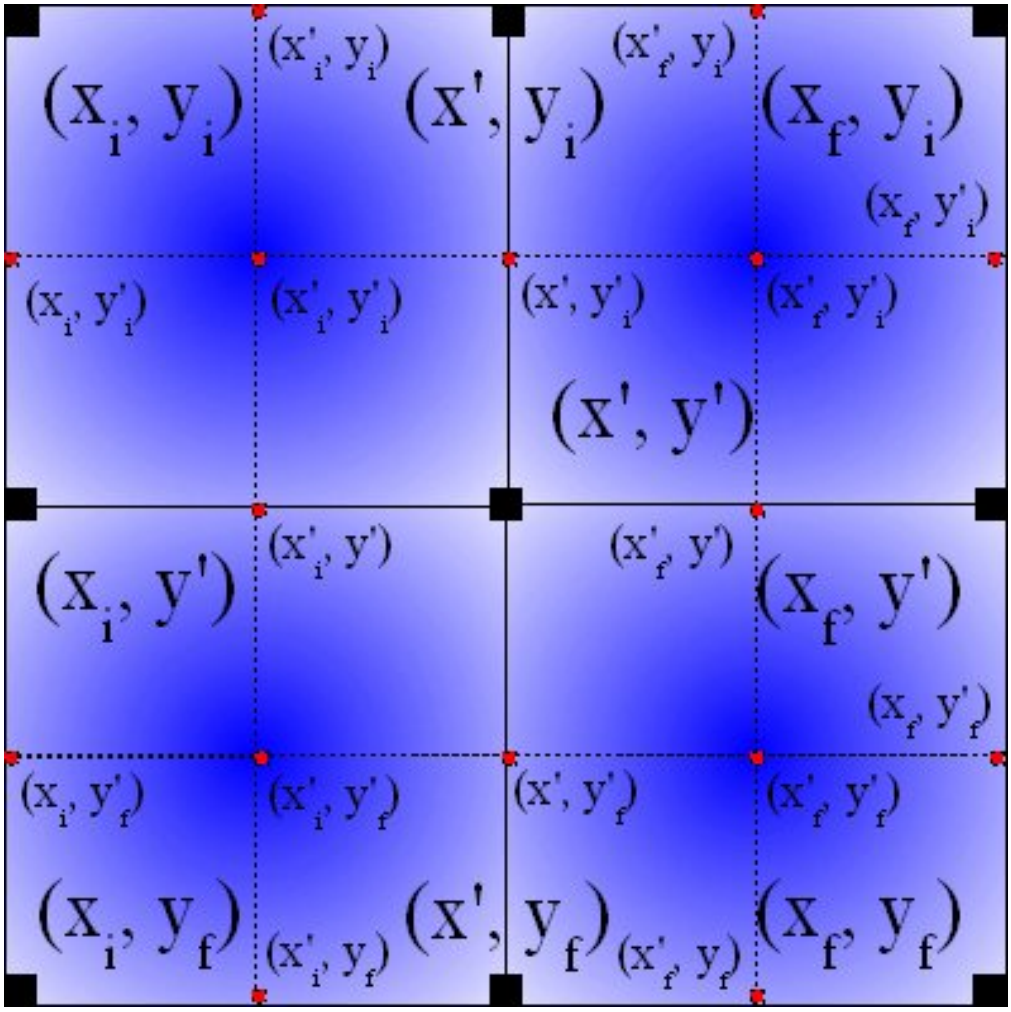


Figura 3 - Siguiente Iteración

Al final del proceso, obtendremos una imagen parecida a la mostrada en la *Figura 4*:

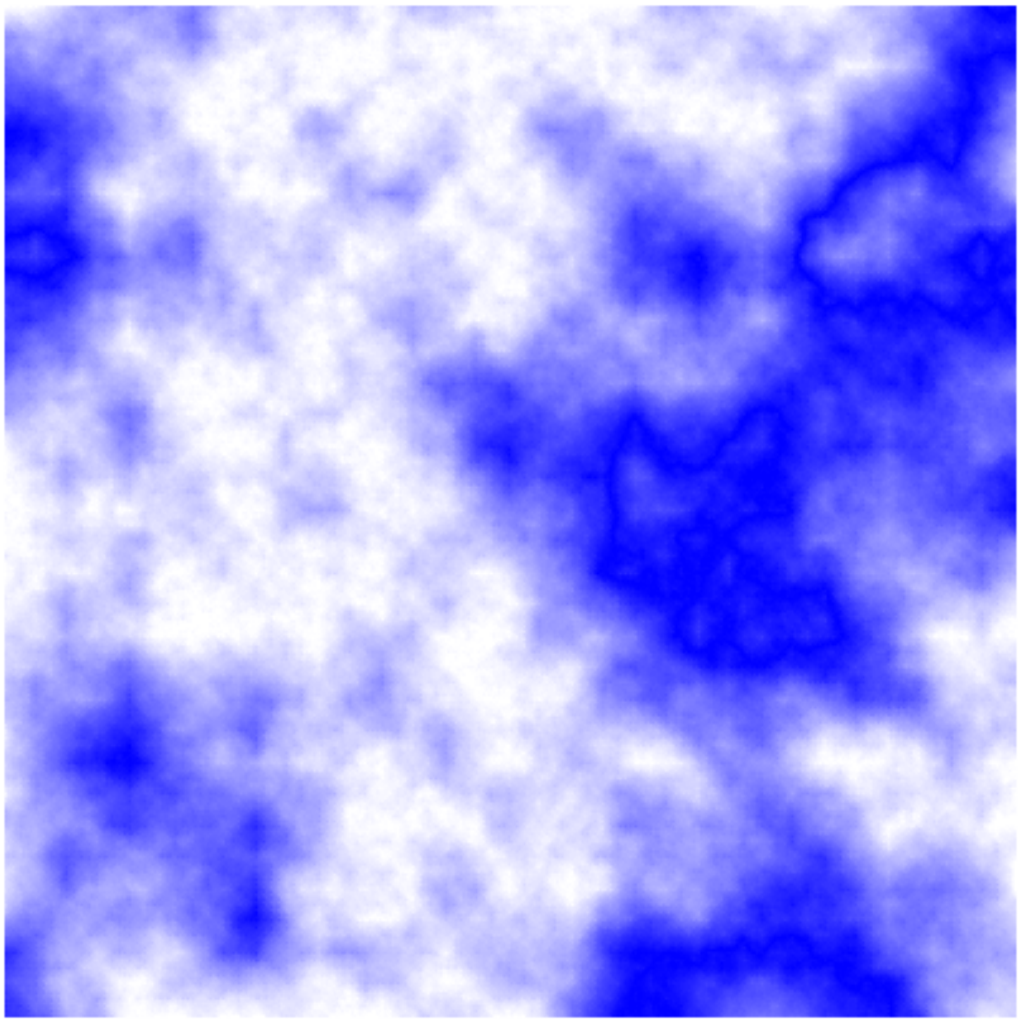


Figura 4

Mapa de Colores

Para elegir un color aleatoriamente, necesitamos crear un mapa de colores. Tal mapa nos permitirá asociar un valor entero a un color determinado. Esto es parecido a una paleta, pero se generará el color *elegido* a partir de una fórmula. El procedimiento para realizar tal mapa de colores es algo sencillo. Basta con comprobar el intervalo del valor entero dado, y generar el color que queremos. Cada vez que queramos un color determinado, simplemente invocamos el mapa de colores para generarlo y obtenerlo. Por ejemplo,

```
COLOR Mapa( i )
1.  Si  i ≤ 0,10
2.    color <- de BLANCO (i=0,00) a AZUL (i=0,10)
3.  Si  0,10 < i ≤ 0,50
4.    color <- de AZUL (i=0,10) a ROJO (i=0,50)
5.  Si  0,50 < i
6.    color <- de ROJO (i=0,50) a AMARILLO (i=1,00)
7.  Terminar( color )
```

El valor usado por el mapa de colores puede ser de cualquier intervalo. Podemos usar porcentajes como valores en el intervalo de [0, 255]. En el algoritmo anterior, obtenemos un color diferente según el porcentaje indicado por *i*. Por ejemplo, si queremos averiguar el color de *i=0,75*, podemos comprobar que se generará un color a mitad de camino entre el rojo y el amarillo, que será naranja, seguramente. Esto es porque 0,75 es la media entre 0,50 y 1,00. Usando el applet presentado anteriormente, conseguiremos un color anaranjado con R=255, V=127, A=0. Este color está a la mitad entre el color rojo: R=255, V=0, A=0 y el color amarillo: R=255, V=255, A=0. Calculando la media de cada intensidad: R=(255+255)/2, V=(0+255)/2, A=(0+0)/2, conseguimos nuestro color naranja: R=255, V=127, A=0.

Para obtener el cálculo anterior, tenemos que crear una fórmula basada en el valor de *i*. Tal fórmula puede ser:

```
6.1.  color.rojo  = 255
6.2.  color.verde = 255*(2*(i-0,50)) = 510*(i-0,50)
6.3.  color.azul  = 0
```

Cuando *i* valga 0,50, *color.verde* será 0 y cuando *i* valga 1,00, *color.verde* será 255. Esto es justamente lo que queremos, ya que podemos generar los demás colores comprendidos entre tales colores.

Algoritmo

El algoritmo se compone de varias funciones importantes:

- Necesitamos la función principal, *Generar_Nube()*, que establecerá los valores iniciales y preparará el procedimiento adecuadamente.
- La función *Mapa()* sirve para generar un color para las nubes a partir de un valor numérico entero no negativo en el intervalo [0, 255].
- La función *Subdividir()* se encargará de la parte recursiva del algoritmo.
- La función *Generar_Parámetro()* se encarga de obtener el color del punto medio de dos puntos. Se calcula la media de los colores de cada punto agregando un valor aleatorio. Como los colores nuevos deben pertenecer a nuestro mapa de colores, se basará el cálculo en los parámetros de los colores, en lugar de los colores verdaderos. Es decir, calcularemos el parámetro que representa el nuevo color. Luego, usaremos *Mapa()* para obtener el color verdadero según el parámetro.

Para *Generar_Nube()*, el algoritmo es:

```
1.  Iniciar: valor_semilla, x_pi, y_pi, x_pf, y_pf
2.  anchura_p <- |x_pf - x_pi + 1|
3.  altura_p  <- |y_pf - y_pi + 1|
```

```
4.  Inicializar: parámetros[anchura_p, altura_p] <- -1 o cualquier
valor negativo
5.  Iniciar_Generador_Números_Aleatorios( valor_semilla )
    // Obtener parámetros de colores
6.  parámetros[0,0] <- Generar_Número(256)
7.  parámetros[anchura_p-1,0] <- Generar_Número(256)
8.  parámetros[anchura_p-1,altura_p-1] <- Generar_Número(256)
9.  parámetros[0,altura_p-1] <- Generar_Número(256)
    // Dibujar Esquinas
10. PonPixel( 0,0, Mapa( parámetros[0,0] ) )
11. PonPixel( anchura_p-1,0, Mapa( parámetros[anchura_p-1,0] ) )
8.  PonPixel( anchura_p-1,altura_p-1, Mapa( parámetros[anchura_p-1,
altura_p-1] ) )
9.  PonPixel( 0,altura_p-1, Mapa( parámetros[0,altura_p-1] ) )
    // Comenzar la Recursividad
10. Subdividir( parámetros, 0,0, anchura_p-1,altura_p-1 )
11. Terminar
```

- La matriz o tabla *parámetros*, de dimensiones $anchura_p$ x $altura_p$, guardará los parámetros usados para obtener un color del mapa, para cada píxel. Inicialmente, esta matriz contendrá valores negativos.
- La función *Rellenar_Imagen(Color)* hace alusión a una función de la librería o API gráfica para asignar un mismo color para cada píxel de la imagen. Es posible que en tal librería gráfica se pueda rellenar la imagen con un color de fondo, previamente establecido. Si es así, entonces *Rellenar_Imagen(Color)* se limita a establecer el color dado como el color de fondo y asegurarse de que la imagen se rellene completamente con tal color.
- Las funciones *Iniciar_Generador_Números_Aleatorios()* y *Generar_Número()* representan funciones estándares que tiene el compilador para generar número aleatorios. La variable *valor_semilla* contiene el valor inicial para comenzar el generador de números pseudo-aleatorios. Usando un valor inicial conocido, podemos recrear la misma imagen con sólo usar el mismo número. De esta forma, asociamos un único valor con cada imagen generada. Para *Generar_Número(N)*, se generará un número aleatoriamente en el intervalo [0, N-1].
- La función *PonPixel()* hace referencia a una función de la librería o API gráfica para establecer un color para un píxel determinado en la pantalla, según las coordenadas dadas.

Para *Mapa()*, el algoritmo es:

```
Color Mapa( x )
1.  Color.rojo  <- |2*x-255|
2.  Color.verde <- |2*x-255|
3.  Color.azul  <- 255
4.  Terminar( Color )
```

Color contiene los valores de cada estímulo del modelo RGB de 24 bpp. El mapa para crear nubes se basa en una gama de colores entre blanco, azul y de vuelta a blanco. Esto es, desde (255,255,255) pasando por (0,0,255) hasta llegar otra vez a (255,255,255). Por supuesto, podemos tener un mapa más simple: de azul a blanco, o incluso de blanco a azul. Sin embargo, para generar nubes, queremos colocar estratégicamente el color blanco en las "puntas" de nuestro mapa.

Para *Subdividir()*, el algoritmo es:

```
Subdividir( matriz, x_i, y_i, x_f, y_f )
1.  Si  x_f-x_i ≥ 2  OR  y_f-y_i ≥ 2, entonces
2.     Terminar
    // Calcular el punto medio
3.  x <- (x_i+x_f) / 2
4.  y <- (y_i+y_f) / 2
    // Dibujar los puntos medios
5.  Si matriz[x,y_i] < 0, entonces
6.     Generar_Parámetro( matriz, x_i,y_i, x,y_i, x_f,y_i )
7.     PonPixel( x,y_i, Mapa( matriz[x,y_i] ) )
8.  Si matriz[x_f,y] < 0, entonces
9.     Generar_Parámetro( matriz, x_f,y_i, x_f,y, x_f,y_f )
10.    PonPixel( x_f,y, Mapa( matriz[x_f,y] ) )
11. Si matriz[x,y_f] < 0, entonces
12.    Generar_Parámetro( matriz, x_i,y_f, x,y_f, x_f,y_f )
13.    PonPixel( x,y_f, Mapa( matriz[x,y_f] ) )
14. Si matriz[x_i,y] < 0, entonces
15.    Generar_Parámetro( matriz, x_i,y_i, x_i,y, x_i,y_f )
16.    PonPixel( x_i,y, Mapa( matriz[x_i,y] ) )
    // Dibujar el punto central = la media de 4 colores
17. matriz[x,y] <- ( matriz[x_i,y_i] + matriz[x_f,y_i] + matriz[x_i,y_f]
+ matriz[x_f,y_f] ) / 4
18. PonPixel( x,y, Mapa( matriz[x,y] ) )
    // Recursividad
19. Subdividir( matriz, x_i,y_i, x,y )   // Cuadrante superior
izquierdo
20. Subdividir( matriz, x,y_i, x_f,y )   // Cuadrante superior
derecho
21. Subdividir( matriz, x,y, x_f,y_f )   // Cuadrante inferior
izquierdo
22. Subdividir( matriz, x_i,y, x,y_f )   // Cuadrante inferior
derecho
23. Terminar
```

- Los parámetros x_i, y_i, x_f, y_f contienen los valores de las coordenadas de la esquina superior izquierda (x_i, y_i), y de la esquina inferior derecha (x_f,y_f).
- La función *TraePixel()* hace alusión a una función de la librería o API gráfica para obtener un color para un píxel determinado en la pantalla, según las coordenadas dadas.

Para *Generar_Parámetro()*, el algoritmo es:

```
Generar_Parámetro( parámetros, xa,xb, x,y, ya,yb )
1.  t <- |xa-xb| + |ya-yb|
2.  t <- Generar_Número(2*t) - t + ( parámetros[xa,ya] + parámetros
[xb, yb] + 1 ) / 2
3.  Si t < 0, entonces
4.     t <- 0
5.  Si no, comprueba Si t > 255, entonces
6.     t <- 255
7.  parámetros[x,y] <- t
8.  Terminar
```

Algunos valores en la matriz *parámetros* son guardados. Huelga decir que tales cambios deben existir al terminar la función *Generar_Parámetro()*. En otras palabras, *parámetros* es un dato entrante y saliente.

Observaciones

Este algoritmo se basa en obtener colores a través de una fórmula paramétrica. El parámetro de dicha fórmula o mapa se obtiene al principio a través de una secuencia de números aleatorios. En nuestro algoritmo, nos interesa manipular cada uno de estos parámetros, en lugar de los colores en sí. Esto es análogo a usar una paleta. En nuestro caso, en vez de guardar los colores generados para luego manipular su índice, calculamos los colores a través de un parámetro. Debemos guardar los parámetros usados mediante la recursividad, por lo que necesitamos una tabla homóloga a la imagen y con las mismas dimensiones. Esto es la tabla *parámetros* en nuestro algoritmo. La *figura 5* muestra esta transformación de nuestro parámetro a un color en nuestro mapa:

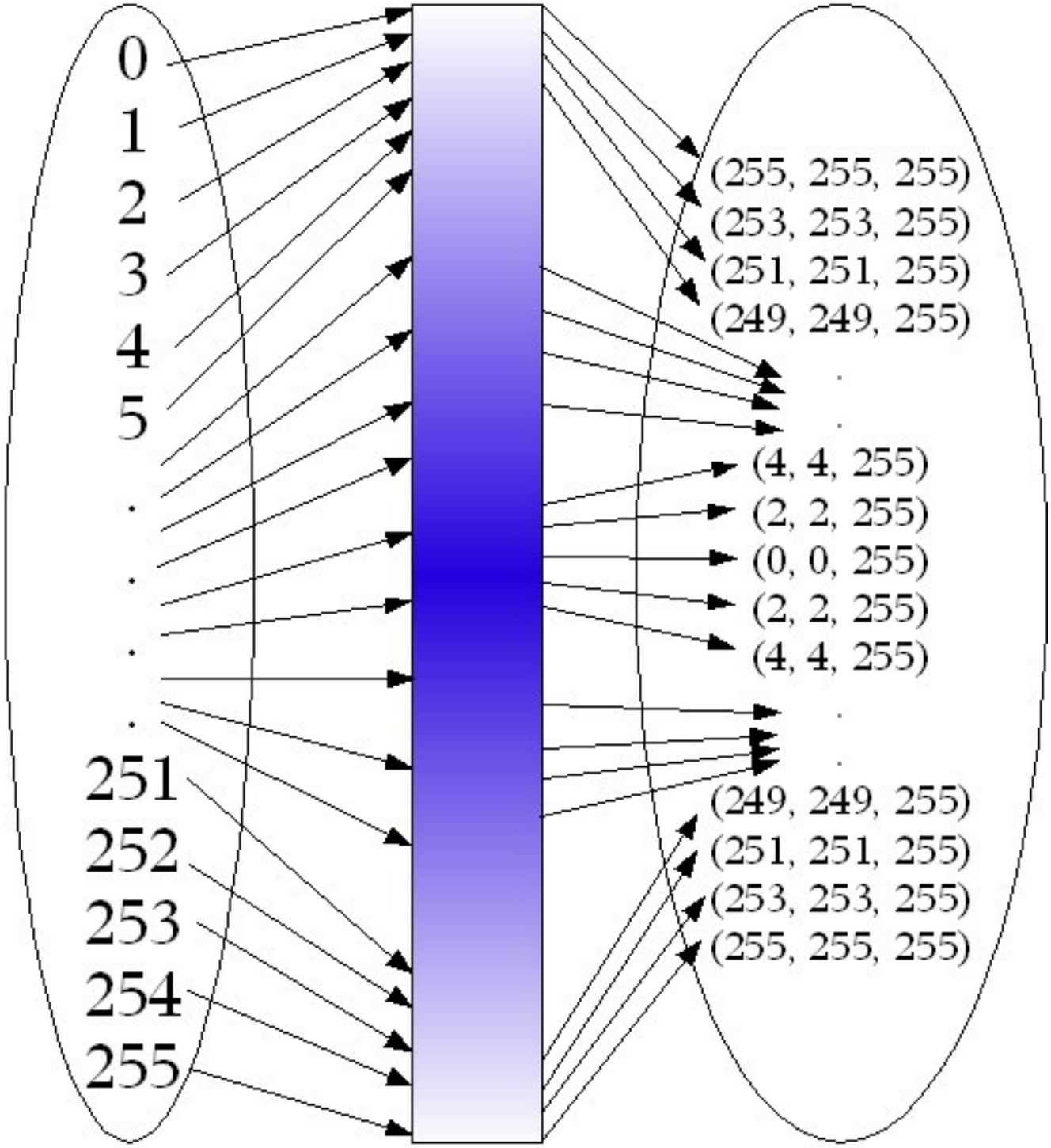


Figura 5 - Mapa: Parámetro-Color

Algunos lectores habrán observado que algunas de las imágenes generadas con este algoritmo tienen una tendencia de formar un rectángulo. Es decir, algunos colores se agrupan de una manera peculiar, creando grupos o brumos con una forma rectangular. Esto se produce debido al método recursivo de nuestro algoritmo, que se basa en calcular los colores de los píxeles de forma rectangular. También es *culpable* el método de perturbación, ya que la variedad es menor, cuanto menor sea la distancia entre los píxeles. Esto implica que los colores nuevos para píxeles cercanos entre sí no varían mucho de los colores de sus vecinos.

Explicación Detallada

Como se ha explicado anteriormente, este algoritmo se basa en un método recursivo y un método de perturbación para obtener un parámetro que luego es usado para calcular un color en nuestro mapa de colores; o sea, para que un color nuevo pertenezca a nuestra gama de colores. Existen otros métodos para generar este tipo de imágenes. El método más popular y usado es el método de Perlin. Este método se basa en crear un mapa semi-infinito de valores pseudo-aleatorios, que sirven para formar varias frecuencias distintas. Tales frecuencias son sumadas entre sí para crear una función continua. Esta función continua tiene como figura estar formado por grandes perturbaciones en algunos puntos aislados, pero muchas perturbaciones pequeñas a lo largo de toda la función. Con esta función, se puede aplicar varios mapas de colores para generar imágenes muy realistas y naturales; desde nubes en un cielo azul (mejores que con el método que hemos usado), pasando por una textura de madera, granito, hasta texturas de tejidos de lana y algodón.

El método de Perlin se usa para generar texturas, pero también se puede aplicar a animación, para simular movimientos con apariencia aleatoria, pero no caótica. Algunos ejemplos de tal uso puede ser el movimiento de las copas de los árboles en una suave brisa o en un vendaval, o incluso el agiteo de alas de una mariposa o pájaro, hasta incluso la simulación de la caída de copos de nieve en una escena navideña.

Tanto las imágenes generadas por nuestro algoritmo, como las del método de Perlin, se denominan *texturas de procedimiento*. Estas texturas de procedimiento son un gran alivio para muchos problemas frecuentes al aplicar imágenes rectangulares a objetos en tres dimensiones. Esto es porque las imágenes precreadas deben ser deformadas para ajustarse a las dimensiones del objeto al igual que una distorsión mayor por motivos de perspectiva. Con texturas de procedimiento, cada color puede ser calculada sin depender de las dimensiones del objeto. Como añado, estas texturas se basan en coordenadas de cualquier dimensión: 1D, 2D, 3D, 4D, etc. y por tanto, no existe una transformación de coordenadas de píxeles a coordenadas del objeto, y de vuelta a píxeles. La propia textura se puede usar directamente en cualquier sistema de coordenadas, sin necesidad de un cambio, o al menos no sería impactante.

Huelga decir que, tanto para nuestro método, como el de Perlin, las ímagenes producidas son algo nítidas o marcadas para el cambio de colores. Por lo tanto, nos conviene aplicar un método de suavizado, para mezclar los colores de la imagen. Esto implica que un conjunto de píxeles de colores idénticos o muy parecidos resultará en el mismo conjunto pero de colores iguales, pero seguramente de menor intensidad. Con esto, podemos eliminar esos colores aislados y poblar más área de la imagen con otros colores mayoritarios. El método de suavizado también se llama *antialiasing*, según los factores que proporcionemos al método.

Trataremos todos estos temas más adelante en el curso, pero aún nos faltan muchos temas y conceptos por tratar.

Ejercicios

Enlace al paquete de este capítulo.

1. Escribir un programa que implemente el algoritmo descrito en este capítulo para generar nubes. Usar una resolución de 400x400.

2. Escribir un programa usando el mapa de colores descrito en cada apartado más abajo, con una resolución de 400x400.

a) Plasma:

```
Color Mapa( x )
1.  Si 0 ≤ x ≤ 85
2.    Color.rojo  <- 0
3.    Color.verde <- 3*x
4.    Color.azul  <- 3*(86-x)
5.  Si 86 ≤ x ≤ 170
6.    Color.rojo  <- 3*(86-x)
7.    Color.verde <- 3*(171-x)
8.    Color.azul  <- 0
9.  Si 171 ≤ x ≤ 255
10.   Color.rojo  <- 3*(255-x)
11.   Color.verde <- 0
12.   Color.azul  <- 3*(x-172)
13. Terminar( Color )
```

b) Fuego-Hielo:

```
Color Mapa( x )
1.  Color.rojo  <- 255*(1+sen(2*π/256*x))/2
2.  Color.verde <- 255*(1+cos(2*π/256*x))/2
3.  Color.azul  <- x
4.  Terminar( Color )
```

c) Topológico:

```
Color Mapa( x )
1.  Si 0 ≤ x ≤ 31
2.    Color.rojo  <- 64
3.    Color.verde <- 127
4.    Color.azul  <- 192
5.  Si 32 ≤ x ≤ 62
6.    Color.rojo  <- 255
7.    Color.verde <- 127
8.    Color.azul  <- 0
9.  Si 63 ≤ x ≤ 255
```

```
10.   x <- 63*((1-(i-63)/192) + 255*(i-63)/192
11.   Color.rojo  <- 255-x/3
12.   Color.verde <- 255-x/2
13.   Color.azul  <- x
14. Terminar( Color )
```

3. Aunque en el algoritmo descrito en este capítulo haga uso de parámetros entre 0 y 255 (ambos incluidos), podemos usar parámetros entre 0,00 y 1,00 (ambos incluidos) como se explicó en la sección, Mapa de Colores. Ahora podemos explicar una forma sencilla de calcular todos los colores entre dos colores conocidos. Se trata del método de interpolación lineal.

En esta fórmula, queremos seguir una progresión lineal entre (x_i, y_i) hasta (x_f, y_f). Nos interesa averiguar el valor y según un valor x; o sea, nos interesa averiguar (x, y). Por lo tanto, tenemos cuatro valores constantes: $\mathbf{x_i}$, $\mathbf{y_i}$, $\mathbf{x_f}$, $\mathbf{y_f}$; una variable dada: **x**; y un valor a calcular: **y**. He aquí la fórmula:

Real Interpolacion(y_i, y_f, x, x_i, x_f)
1. y <- y_i*(x-x_f)/(x_i-x_f) + y_f*(x-x_i)/(x_f-x_i)
2. Terminar(y)

donde, $x_i \leq x \leq x_f$

Observando esta fórmula, podemos ver que,
Si $x = x_i$, entonces $y = y_i*(x_i-x_f)/(x_i-x_f) + y_f*0 = y_i * 1 = y_i$,
Si $x = x_f$, entonces $y = y_i*0 + y_f*(x_f-x_i)/(x_f-x_i) = y_f * 1 = y_f$

De esta forma, nos "desplazamos" desde (x_i, y_i) hasta (x_f, y_f), encontrando otros valores que siguen una línea recta.

Por ejemplo, tenemos (2, 5) y (4, 10). Queremos averiguar los valores para la pareja (3, y). Aplicando la fórmula, tenemos que,

$$y = 5 * \frac{(3 - 2)}{(4 - 2)} + 10 * \frac{(3 - 4)}{(2 - 4)} = 5/2 + 10/2 = 7,5$$

El valor que buscamos es y = 7,50, por lo que tenemos que (3, 7,50).

Tomando el mapa descrito en el ejemplo de la sección, Mapa de Colores, veamos la forma de convertirlo usando interpolación lineal.

```
COLOR Mapa( i )
1.  Si  i ≤ 0,10
      // De Blanco a Azul
```

```
2.    color.rojo  <- 255*(0,10-i)/0,10
3.    color.verde <- 255*(0,10-i)/0,10
4.    color.azul  <- 255
5.  Si  0,10 < i ≤ 0,50
      // De Azul a Rojo
6.    color.rojo  <- 255*(i-0,10)/0,40
7.    color.verde <- 0
8.    color.azul  <- 255*(i-0,50)/0,40
      // De Rojo a Amarillo
9.  Si  0,50 < i
6.    color.rojo  <- 255*(1,00-i)/0,50
7.    color.verde <- 255*(i-1,00)/0,50
8.    color.azul  <- 255*(i-0,50)/0,40
13. Terminar( color )
```

Crear algunos mapas basados en esta técnica de interpolación.

4. Como se hizo en los ejercicios del capítulo 3d - Fractales, es aconsejable crear varias funciones con cada mapa que se desea usar. Luego, al invocar el proceso, pasamos un puntero a la función con el mapa de colores que nos interesa.

Crear más mapas y seguir el proceso descrito anteriormente, usando punteros a funciones, para aplicar cualquier mapa de colores a nuestras nubes para formar otras imágenes. Se recomienda mapas de colores que sigan una fluidez de un color a otro con suavidad.

5. Por ahora, hemos estado creando mapas de colores directamente con el parámetro calculado. Otra fórmula que podemos implementar es en el trato del parámetro. En lugar de escoger un color de nuestro mapa usando el parámetro, aplicaremos una ecuación al parámetro antes de escoger nuestro color.

a) Aplicar la función del seno al parámetro. Si se usa el parámetro de 0,00 á 1,00, entonces se puede seguir la siguiente fórmula:

```
parámetro <- (1 + sen( 2π*matriz[x,y] )) / 2
PonPíxel( x, y, Mapa( parámetro ) )
```

b) Aplicar la función del exponencial al parámetro. De nuevo, si el parámetro se encuentra en el intervalo: [0,00, 1,00], entonces usar la siguiente fórmula:

```
parámetro <- (exp(matriz[x,y]) - 1) / (e-1)
PonPíxel( x, y, Mapa( parámetro ) )
```

donde, `e = 2,7182818285`

Observación: Aplicando una función exponencial es una técnica de suavizar una imagen. Compara

la imagen de nubes producida con el método descrito en este capítulo y la imagen generada aplicando la función exponencial. En general, notarás una imagen más suave y algo borrosa generada con la función exponencial.

c) Aplicar la función del seno hiperbólico al parámetro. si el parámetro se encuentra en el intervalo: [0,00, 1,00], entonces usar la siguiente fórmula:

```
parámetro <- senh(matriz[x,y]) / senh(1)
PonPíxel( x, y, Mapa( parámetro ) )
```

d) Aplicar la función del coseno hiperbólico al parámetro. si el parámetro se encuentra en el intervalo: [0,00, 1,00], entonces usar la siguiente fórmula:

```
parámetro <- (cosh(matriz[x,y])-1) / (cosh(1)-1)
PonPíxel( x, y, Mapa( parámetro ) )
```

Es posible que en algunas librerías estándares, la implementación de `exp()` no dé buenos resultados. Por lo tanto, usar la siguiente aproximación a `exp()`:

```
double _exp( double x )
{
  return 1.0 + x*(1.0 + x*(0.5 + x*(1.0/6.0 + x*(1.0/24.0 +
x/120.0))));
}
```

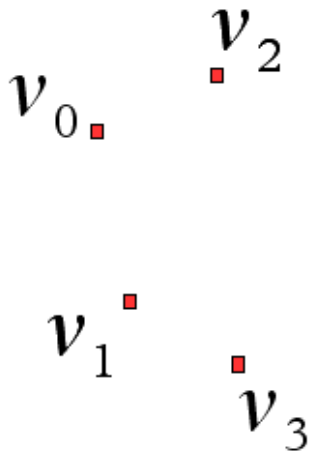
Capítulo 5 - Figuras Geométricas

Uno de los principales objetos o entidades en el campo de la programación gráfica es la figura geométrica. Algunas figuras se usan más que otras, pero todas pueden sernos útiles a la hora de representarlas gráficamente. Algunas API's gráficas contienen todas o algunas de las siguientes figuras geométricas, que trataremos a continuación.

Figuras Geométricas Básicas

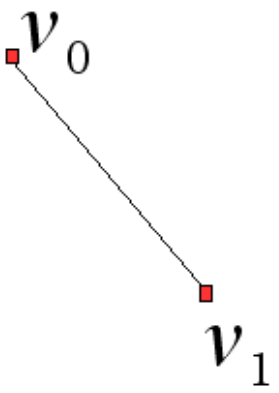
Vértices

No son figuras geométricas como tales, pero forman la base de toda figura geométrica. En algunas ocasiones, nos interesa mostrar los vértices como puntos en la pantalla: v_0, v_1, v_2, ..., v_n.



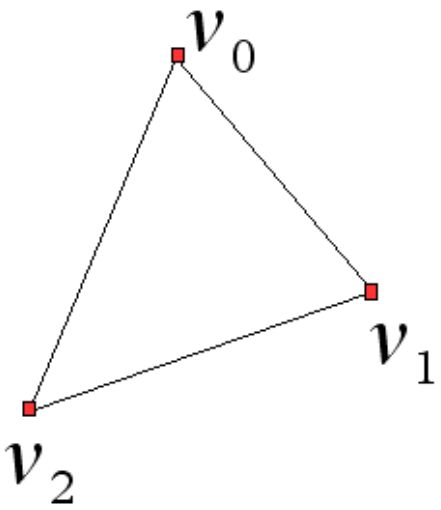
Línea Recta

Matemáticamente hablando, no se trata de una línea recta, sino de un segmento. Aunque no se trate de una figura geométrica, como tal, la línea recta forma la base de los trazados de las figuras. Requerimos dos vértices para crear una línea o segmento: v_0, y v_1.



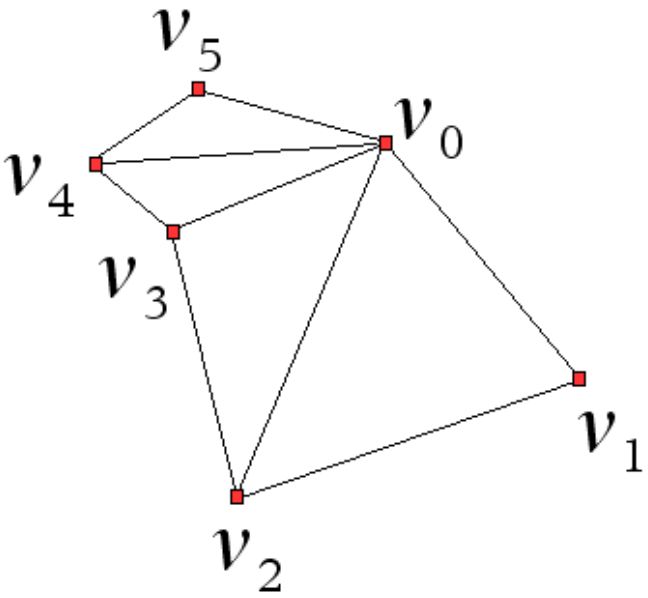
Triángulo

Esta figura geométrica también forma la base de la representación de objetos gráficos, especialmente a la hora de representar objetos en tres dimensiones: v_0, v_1, y v_2. Podemos crear un objeto complejo a partir de triángulos. Necesitamos tres vértices para formar un triángulo.



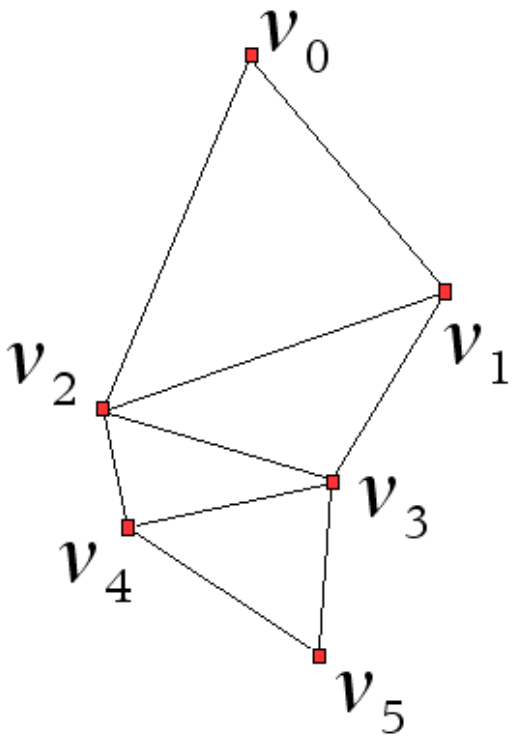
Abanico de Triángulos

Se trata de una serie de triángulos unidos entre sí con un vértice común como el centro del abanico. Como mínimo debe haber tres vértices para formar el primer triángulo: v_0, v_1, y v_2. Los siguientes triángulos se crean a partir de un vértice nuevo, el vértice central del abanico, y el vértice perteneciente al triángulo contiguo.



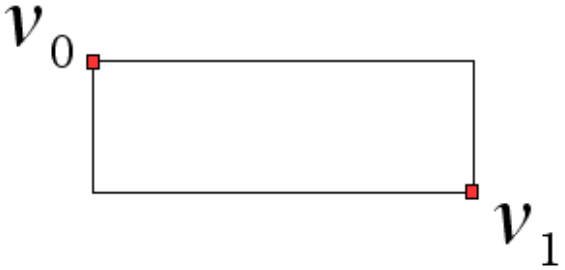
Tira de Triángulos

Se trata de otra serie de triángulos unidos entre sí para formar un cinta o tira. Como mínimo debe haber tres vértices para formar el triángulo inicial: v_0, v_1, y v_2. Los siguientes triángulos se forman a partir de un vértice nuevo y dos vértices que pertenecen al triángulo contiguo.



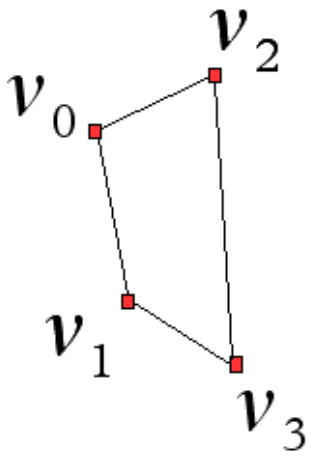
Rectángulo

Otra figura geométrica importante es el rectángulo que también se usa para dibujar cuadrados. Debido a su simetría, sólo requerimos conocer dos vértices, v_0, y v_1, que forman dos esquinas opuestas. En total, obtendremos cuatro números diferentes, con los cuales podemos averiguar las coordenadas de los otros dos vértices.



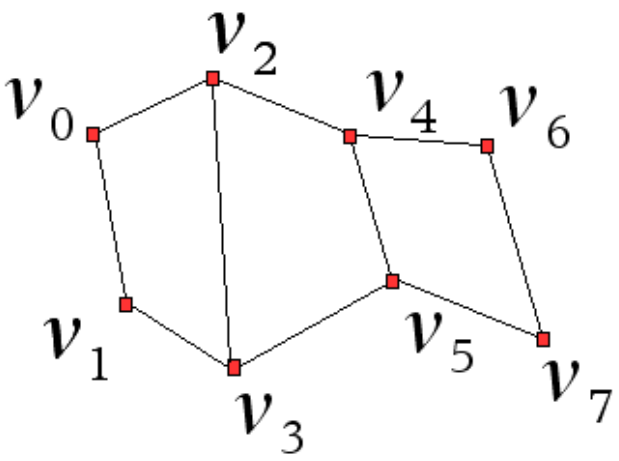
Cuadrilátero

Esta figura geométrica también es usada con frecuencia, tanto para representar objetos en dos como en tres dimensiones. Necesitamos cuatro vértices para formar el cuadrilátero, v_0, v_1, v_2, y v_3.



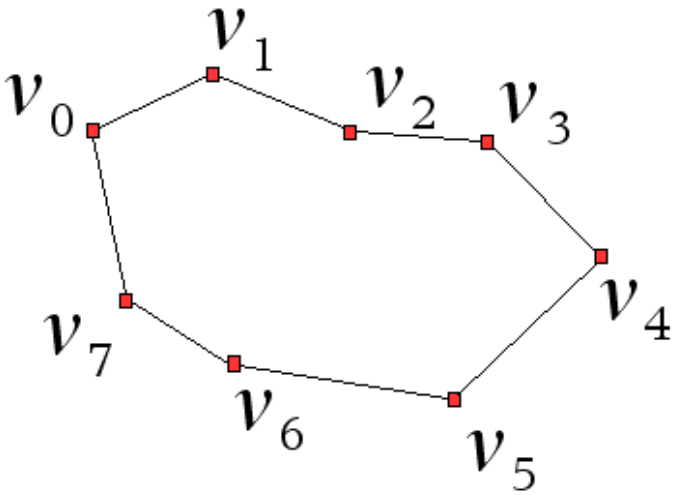
Tira de Cuadriláteros

Se trata de una serie de cuadriláteros unidos entre sí para formar un cinta o tira. Como mínimo necesitamos cuatro vértices para el cuadrilátero inicial: v_0, v_1, v_2, y v_3. Los siguientes cuadriláteros, en la tira, se forman con dos vértices nuevos y con dos vértices pertenecientes al cuadrilátero contiguo.



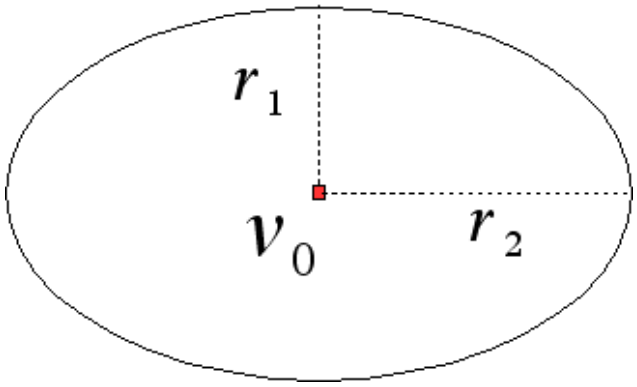
Polígonos

Generalmente, las figuras que nos interesan suelen ser irregulares y con muchos más vértices que 2, 3, ó 4. Además, algunas API's gráficas ofrecen la funcionalidad de describir figuras geométricas según una lista de vértices. Típicamente, la funcionalidad de crear polígonos requiere *n* vértices, donde automáticamente se une el último vértice, v_n, con el primero, v_0, para poder cerrar la figura y formar un polígono [cerrado].

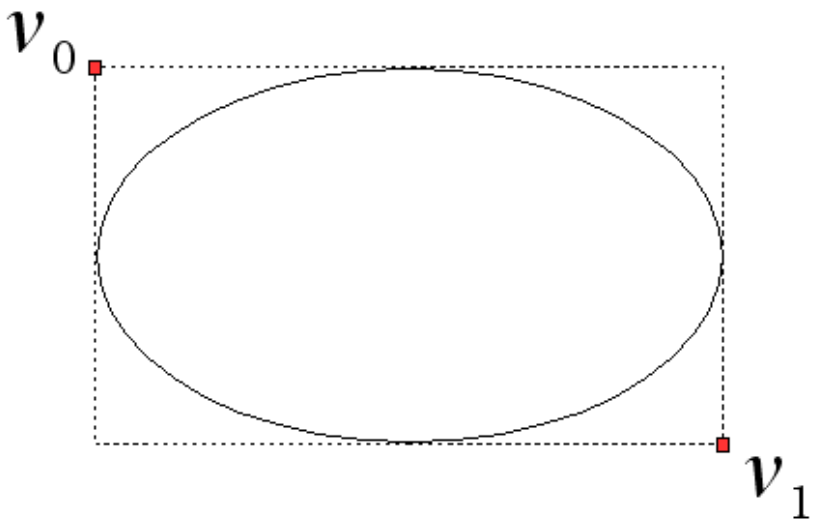


Elipse/Círculo

La elipse al igual que el círculo son otras figuras que nos pueden servir en ocasiones, dependiendo de la escena o "dibujo" que queramos crear. El círculo, o la circunferencia, si la preferimos, es un caso especial de la elipse. Algunos API's gráficos describen las elipses y círculos de formas diferentes. Existen principalmente dos maneras de describir tales figuras:

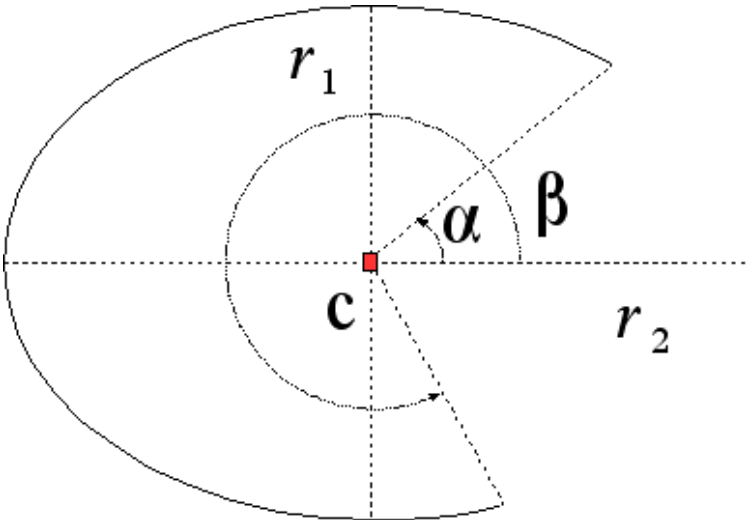


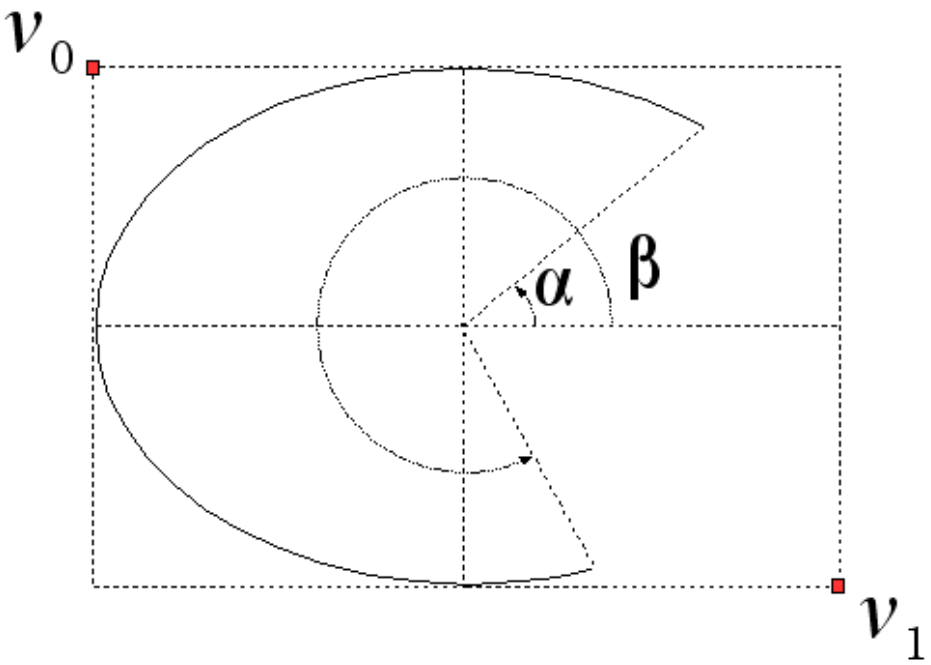
1. Punto central, *c*; y dos longitudes para ambos radios, r_1 y r_2; o
2. Un rectángulo, v_0 y v_1, que circunscribe la elipse o el círculo.



Arco

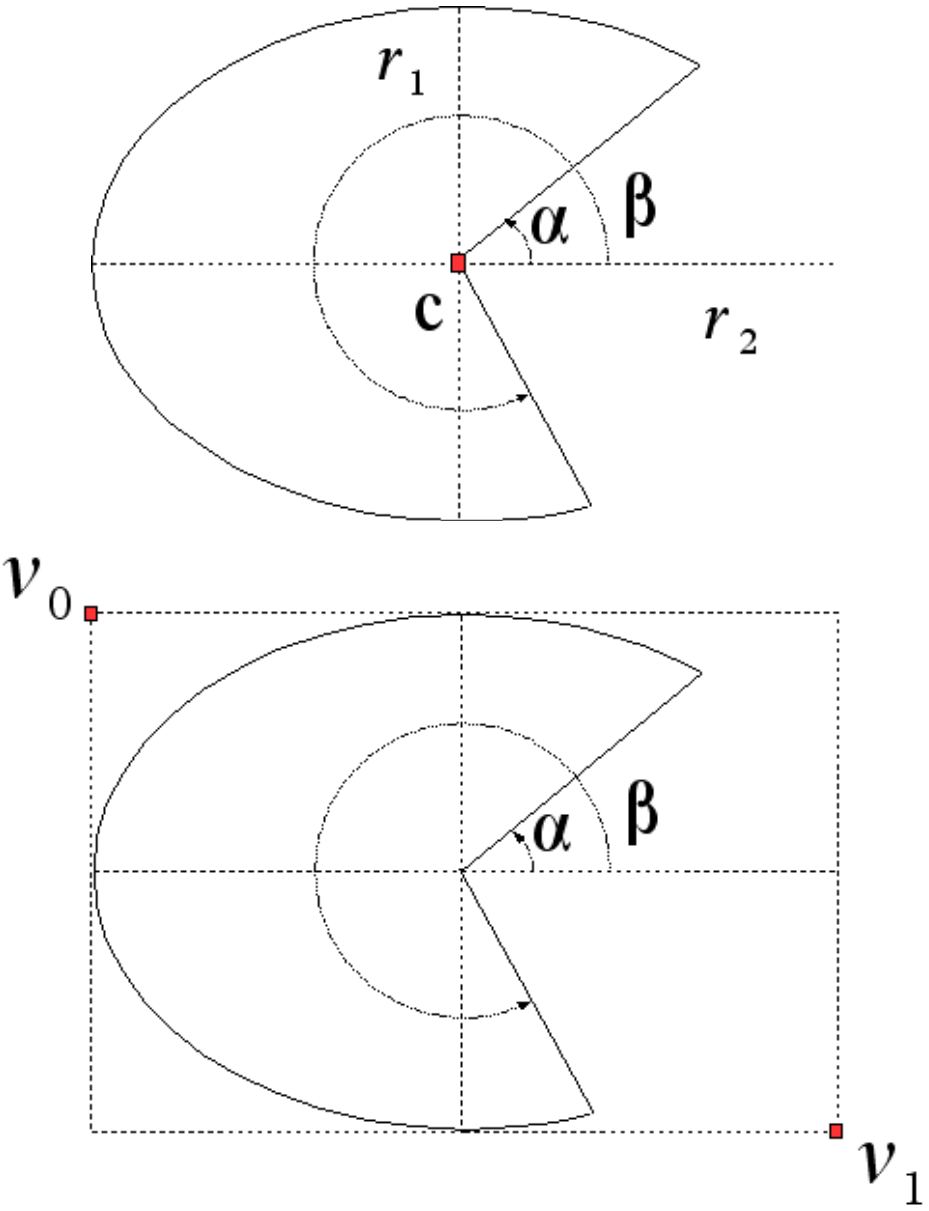
El arco es una línea curva que describe el perímetro total o parcial de una elipse o circunferencia. Como se trata de una línea, no podemos hablar de un área ni rellenarla. La mayoría de las API's gráficas ofrecen tal elemento gráfico con una de dos formas para su descripción:
1. Punto central, *c*; dos radios, r_1 y r_2; y dos ángulos, para describir el ángulo inicial, α, y el final, β; o
2. Un rectángulo, v_0 y v_1, que circunscribe el arco; y dos ángulos, para describir el ángulo inicial, α, y el final, β.





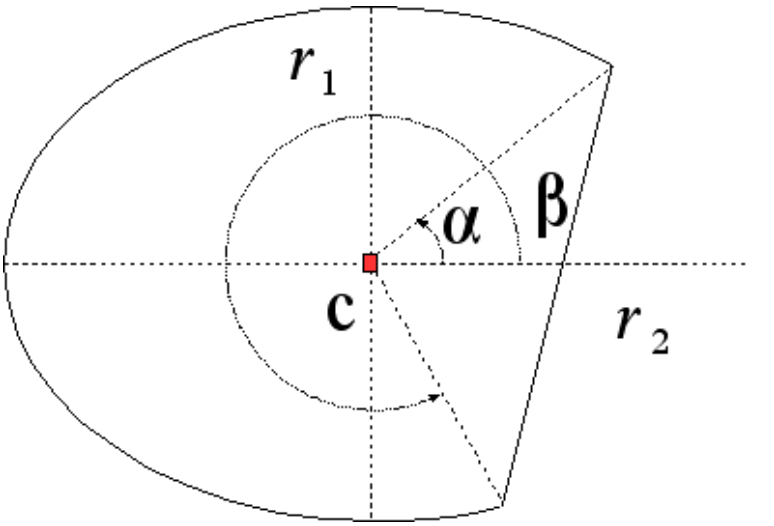
Sector/Cuña

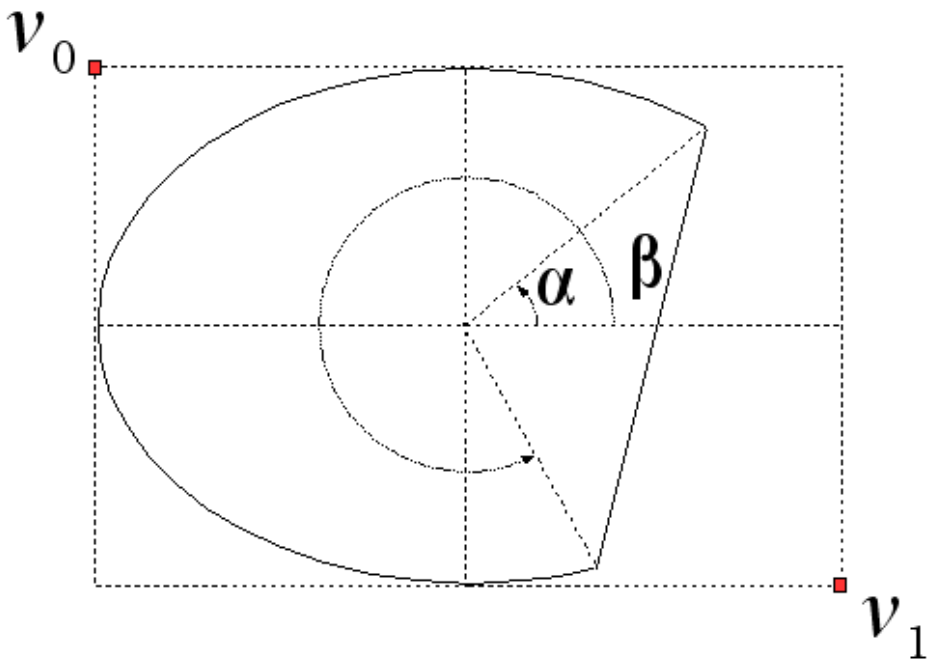
Podemos crear cuñas o sectores que básicamente se parecen a un "trozo de tarta". Esta superficie se basa en una elipse, como el arco, mencionado previamente. Creamos el sector como el arco, pero uniendo cada extremo al centro de la elipse o círculo con un segmento. Existen dos formas populares para describir un sector:
1. Punto central, *c*; dos radios, r_1 y r_2; y dos ángulos, para describir el ángulo inicial, α, y el final, β; o
2. Un rectángulo, v_0 y v_1, que circunscribe el sector; y dos ángulos, para describir el ángulo inicial, α, y el final, β.



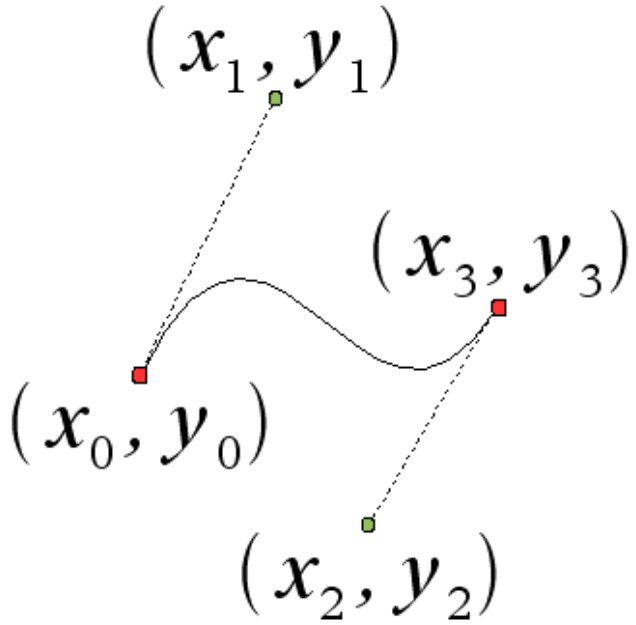
Segmento elíptico

El segmento elíptico se basa en una elipse como el arco y el sector. Esta figura se parece a un sector, pero en lugar de unir cada extremo al centro con segmentos, unimos los extremos entre sí con un segmento. Existen dos formas comunes para describir un segmento elíptico:
1. Punto central, *c*; dos radios, r_1 y r_2; y dos ángulos, para describir el ángulo inicial, α, y el final, β; o
2. Un rectángulo, v_0 y v_1, que circunscribe el segmento elíptico; y dos ángulos, para describir el ángulo inicial, α, y el final, β.





Curva de Bézier



Esta curva fue descubierta por Pierre Bézier, mientras trabajaba para la compañía de automóviles, Renault, en los años 1960. Para poder diseñar adecuadamente las superficies curvas de los automóviles modernos, necesitaba una forma matemática, pero sencilla. Encontró la solución con una ecuación paramétrica definida como ecuaciones cúbicas. Por esta razón, también se denomina esta curva como *curva de Bézier del tercer orden*, al usar una ecuación cúbica. Se describe esta curva con cuatro puntos: dos puntos indican el punto inicial, (x_0, y_0), y final, (x_3, y_3), de la curva, mientras que los otros dos sirven como puntos de control: (x_1, y_1) y (x_2, y_2). Estos puntos de control modifican la curva dependiendo de su posición relativa a los puntos iniciales y finales. Se puede pensar en estos puntos de control como si describieran una atracción, como la gravedad, tirando parte de la curva hacia su centro.

Las fórmulas se basan en el parámetro *t* que queda comprendido en el intervalo [0,1]. Éstas son:

$$x(t) = A_x t^3 + B_x t^2 + C_x t + x_0$$
$$x_1 = x_0 + C_x / 3$$
$$x_2 = x_1 + (C_x + B_x) / 3$$
$$x_3 = x_0 + C_x + B_x + A_x$$
$$y(t) = A_y t^3 + B_y t^2 + C_y t + y_0$$

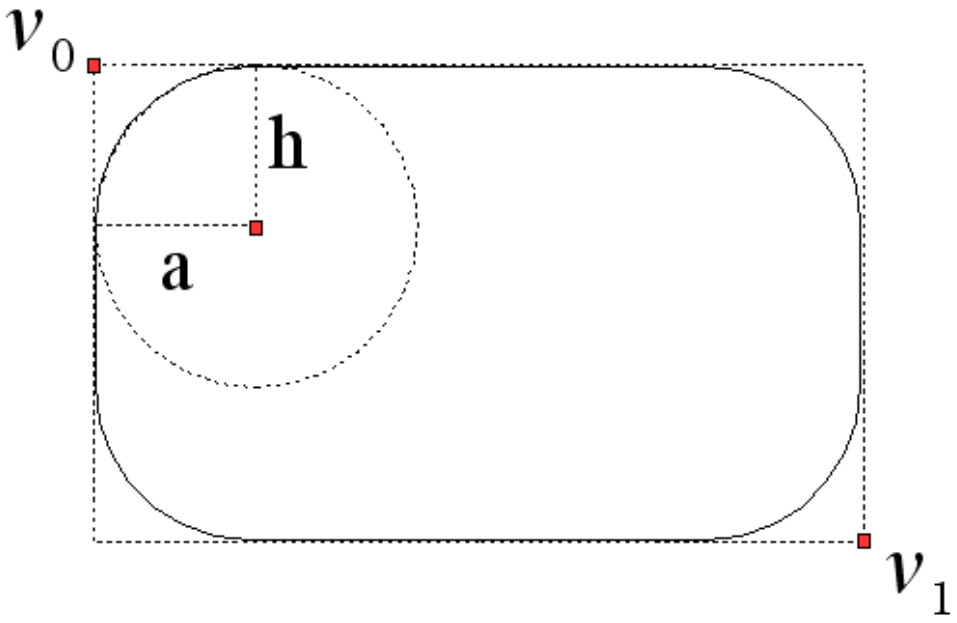
$$y_1 = y_0 + C_y / 3$$
$$y_2 = y_1 + (C_y + B_y) / 3$$
$$y_3 = y_0 + C_y + B_y + A_y$$

Con estas ecuaciones, podemos calcular los respectivos coeficientes para ***x(t)***, e ***y (t)***:

$$C_x = 3*(x_1 - x_0)$$
$$B_x = 3*(x_2 - x_1) - C_x$$
$$A_x = x_3 - x_0 - C_x - B_x$$
$$C_y = 3*(y_1 - y_0)$$
$$B_y = 3*(y_2 - y_1) - C_y$$
$$A_y = y_3 - y_0 - C_y - B_y$$

Rectángulo Redondo

Una variante del rectángulo es el "rectángulo redondo". Esta figura es similar a un rectángulo, pero las cuatro esquinas son curvas. Las curvas son cuartos de un círculo; o sea, 90° de un círculo. Podemos describir esta figura con dos vértices, v_0, y v_1, que forman las dos esquinas opuestas del rectángulo. Luego, necesitamos otros dos valores que indican la altura, *h*, y la anchura, *a*. Ambos valores describen el centro de la circunferencia con respecto a las esquinas del rectángulo. Tal información afecta todas las esquinas, por lo que el cuarto de círculo es la misma figura para cada una.



Ejemplos de Figuras Geométricas

Veamos algunos ejemplos de otras figuras geométricas que podemos construir a partir de las figuras fundamentales que hemos descrito anteriormente.

1. Polígonos Regulares

Un polígono regular se compone de aristas/lados de igual longitud. Esto implica que el ángulo entre cada arista contigua es el mismo. Si trazamos un segmento del centro a un vértice y otro segmento del centro a otro vértice contiguo, entonces el ángulo entre estos dos segmentos es un divisor de $2\pi = 360°$. En otras palabras, cada ángulo mencionado es inversamente proporcional a la cantidad de lados del polígono regular. Podemos usar la siguiente fórmula:

$$\alpha = 2\pi / N, \quad \text{donde } \alpha \text{ es el ángulo, y } N \text{ es la cantidad de lados}$$

Crearemos polígonos regulares en base a una circunferencia que circunscribe nuestro polígono regular. Esto implica, que el centro de la circunferencia coincide con el centro geométrico de cualquier polígono regular. Para esto, necesitamos usar algunas funciones trigonométricas, junto con el ángulo ya calculado. El paso principal es averiguar la coordenada del siguiente vértice de nuestro polígono. Usaremos las siguientes fórmulas:

$$x_i = c_x + r * \cos(i*\alpha),$$
$$y_i = c_y + r * \text{sen}(i*\alpha)$$

donde:
i = 0,1,2,...,N-1,
r es el radio de la circunferencia, y
c = (c_x, c_y) es la coordenada del centro geométrico de la circunferencia y del polígono.

Al agregar el centro a nuestra fórmula, conseguimos *mover* el centro geométrico del origen (0,0) al que nosotros deseemos.

1.1. Pentágono Regular

Con un pentágono, tenemos que N=5, por lo que $\alpha = 2\pi / 5$, que equivale a 72°. Suponiendo que c = (0,0) y r = 1,00, la lista de vértices es:
(1,0000, 0,0000),
(0,3090, 0,9511),
(-0,8090, 0,5878),
(-0,8090, -0,5878),
(0,3090, -0,9511)

Podemos ver el resultado en la figura 16.

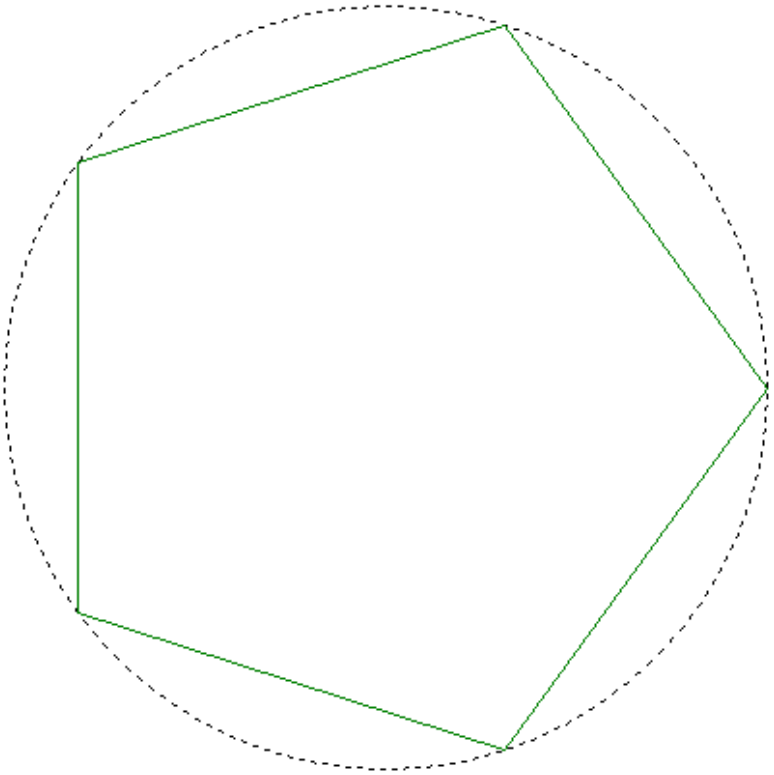


Figura 16 - Polígono - N=5

1.2. Algoritmo

Ahora presentaremos el algoritmo para implementar el trazado de un polígono regular:

```
Polígono_Regular( N, centro, radio )
1.  Crear una lista de N vértices: vértices
2.  alfa <- 2π / N
3.  Bucle: i <- 1 hasta N
4.     vértices[i].x <- centro.x + radio * cos( (i-1)*alfa )
```

```
5.     vértices[i].y <- centro.y + radio * sen( (i-1)*alfa )
6.  Dibujar_Polígono( vértices, N )
7.  Terminar.
```

La función *Dibujar_Polígono()* se refiere a la función de la librería/biblioteca o API gráfica para trazar líneas rectas consecutivas de un vértice o punto a otro según una lista de puntos. Si tal función primitiva no existe, entonces a continuación presentaremos su algoritmo usando la función primitiva, *Dibujar_Línea()*:

```
Dibujar_Polígono( vértices, N )
1.  Bucle: i <- 2 hasta N
2.     Dibujar_Línea( vértices[i-1].x, vértices[i-1].y, vértices[i].x, vértices[i].y )
3.  Dibujar_Línea( vértices[N].x, vértices[N].y, vértices[1].x, vértices[1].y )
4.  Terminar.
```

2. Composición

El objetivo de usar estas figuras geométricas es para poder construir figuras más complejas basadas en las primitivas. He aquí algunos ejemplos de figuras compuestas.

2.1. Cara

Un ejemplo relativamente sencillo de una imagen compleja es crear una cara. Para no complicar el ejemplo, dibujaremos la cabeza como un círculo, los dos ojos como dos círculos, la nariz como un triángulo, y la boca como un semicírculo. El resultado se puede ver en la figura 17.

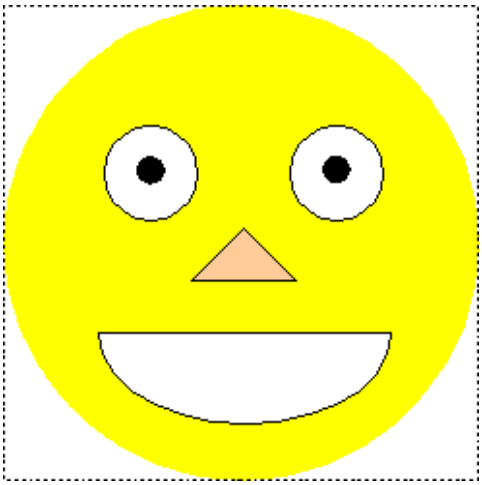


Figura 17 - Cara

2.2. Arco Persa

Otro ejemplo es el de dibujar una ventana de estilo persa. Esto consiste en dibujar varios arcos, unidos entre sí, seguidos de líneas rectas. La figura 18 muestra una simple imagen de la idea citada.

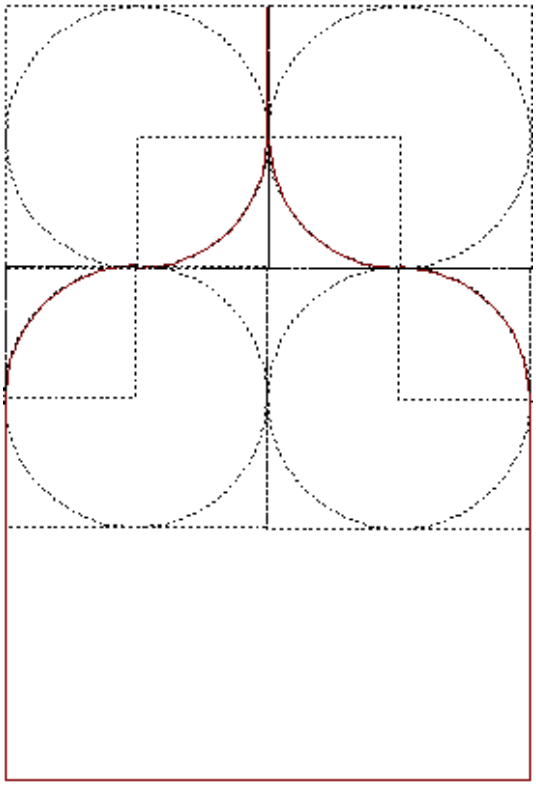


Figura 18 - Arco Persa

3. Visualización de Datos

Una aplicación en asuntos laborales, pero menos artística, es en la presentación de datos con ámbito visual. Desde la presentación del censo popular de una región usando barras hasta las presentaciones de cifras de precios de varios productos en una empresa usando diagramas en tartas (pie charts, en inglés) hacen uso de gráficos simples pero potentes en cuanto a la información que conllevan. Aquí presentaremos algunos ejemplos de tales usos.

3.1. Barras

Podríamos decir que las barras son el gráfico architípico de las presentaciones en empresas al igual que los estadísticos al mostrar las diferencias en población. La figura 19 muestra un claro ejemplo de tales datos.

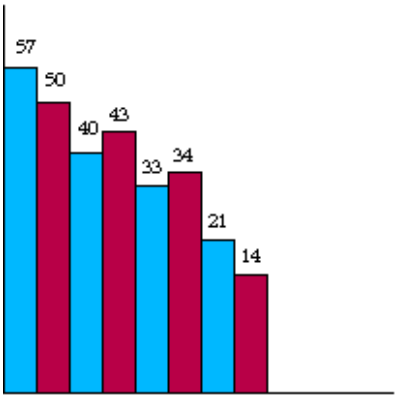


Figura 19 - Barras

3.2. Gráficos circulares

Otro gráfico popular en la representación de estadísticas es la "tarta" o gráfico circular. La figura 20 nos da una idea del conocido gráfico.

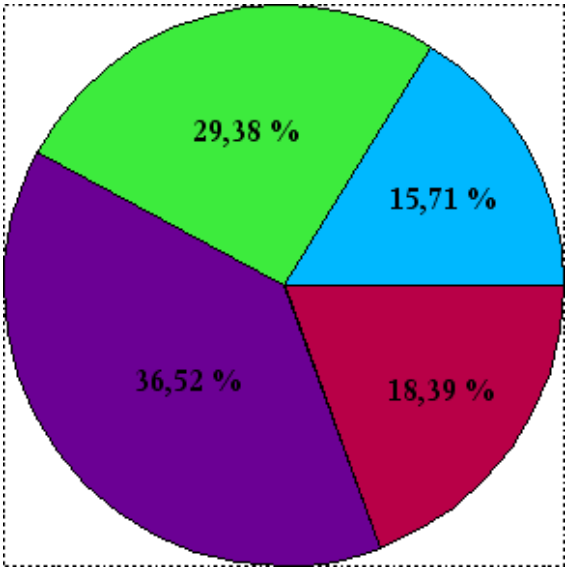


Figura 20 - Gráfico Circular

4. Diagramas

Otro ejemplo con aplicaciones, principalmente, en diseño es en presentar diagramas. Aquí mostramos, en la figura 21, el diagrama de un circuito eléctrico.

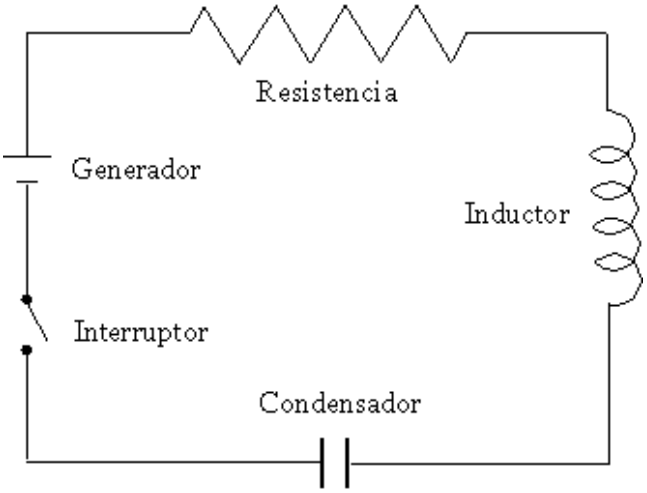


Figura 21 - Circuito Eléctrico

5. Interfaces Gráficas

En el tema de entornos gráficos, un claro ejemplo de las figuras geométricas básicas que hemos visto consiste en la creación de componentes visuales para interfaces gráficas. Por ejemplo, en entornos visuales como MS-Windows®, existen componentes como botones, ventanas, menúes, pestañas, barras de proceso, listados, etc.. Internamente, tales componentes se basan en figuras fundamentales como rectángulos, círculos, y líneas rectas. La figura 22 muestra la típica ventana en MS-Windows®.

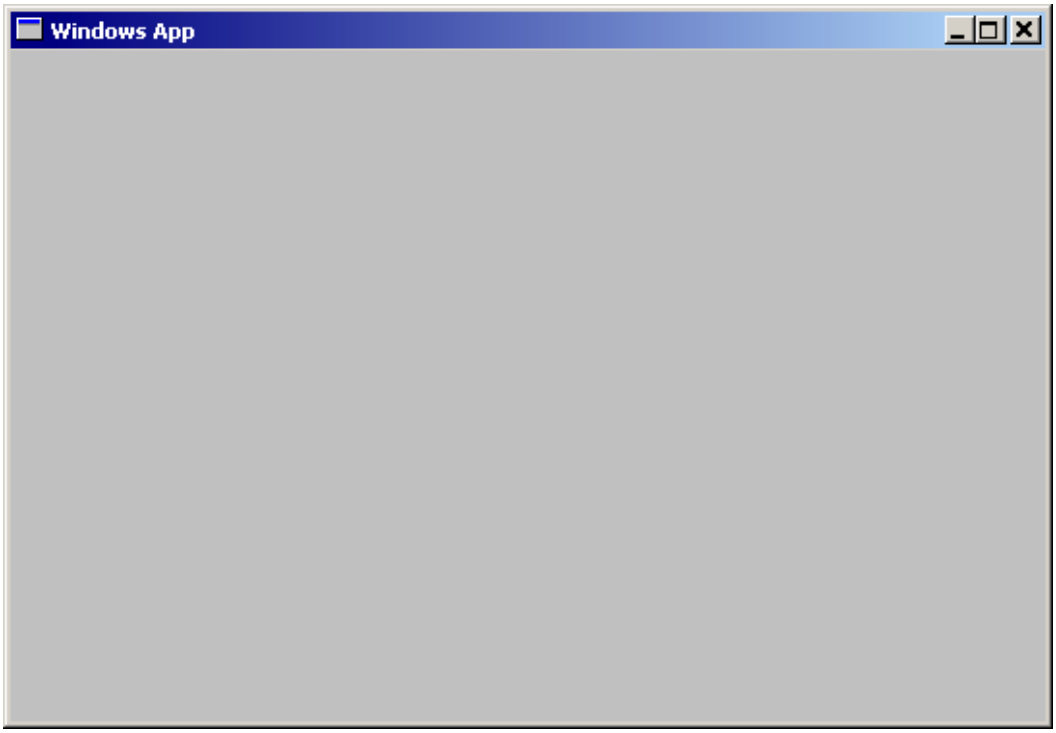


Figura 22 - Ventana de MS-Windows

6. Curvas

Ya hemos visto las curvas de Bézier, mencionando las aplicaciones en el diseño de automóviles. En la programación gráfica, estas curvas son aplicadas popularmente para realizar logotipos, fuentes, y otras figuras que no se pueden crear con facilidad como composiciones de figuras básicas. En la siguiente figura 23, mostramos un fantasma diseñado con varias curvas de Bézier.

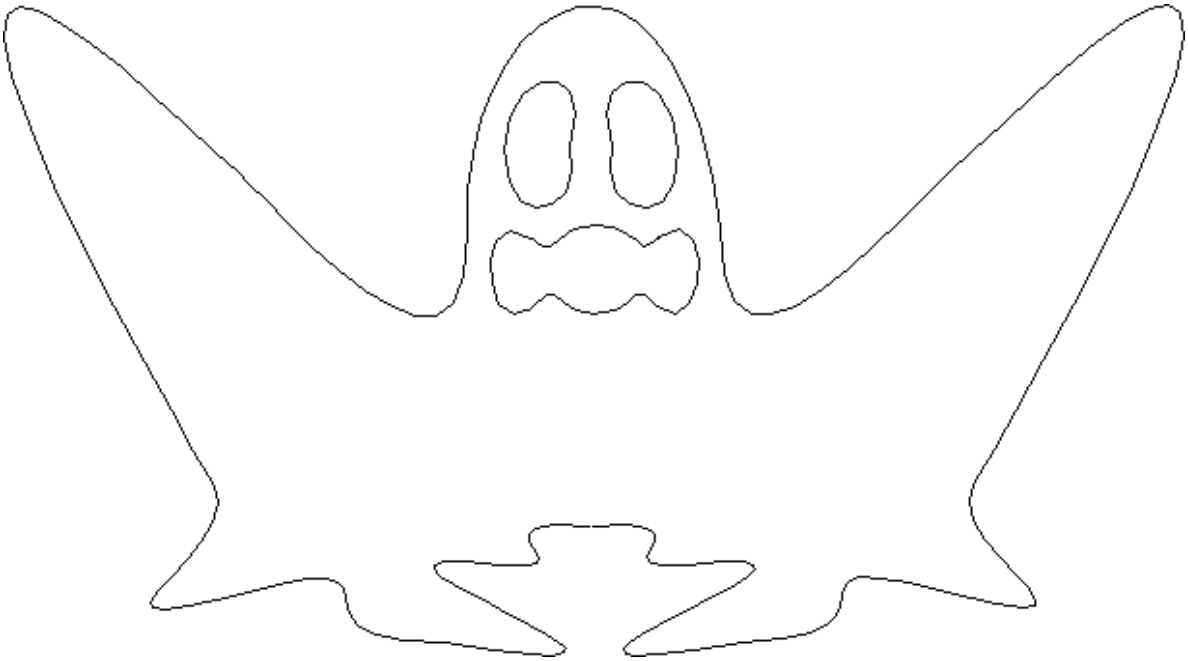


Figura 23 - Fantasma (Bézier)

Para facilitar a los lectores el diseño de las curvas de Bézier, presentamos un applet de Java para poder diseñar sus propias curvas interactivamente. Pinche el vértice que se desee mover con el botón izquierdo del ratón. Mantenga pulsado este botón izquierdo y mueva el ratón en la dirección que se desee, para mudar el vértice de la curva. También se permite introducir las coordenadas directamente a través de los cuadros de edición correspondientes a cada uno de los vértices.

Applet para Curvas de Bézier

Nota: Se requiere la máquna virtual de Sun (Sun VM) para poder ejecutar este applet de Java.

Ejercicios

Enlace al paquete de este capítulo.

1. Escribir un programa que muestre cualquier polígono regular, como se ha descrito en este capítulo. Intentar con los siguientes parámetros:
a) N = 7
b) N = 10
c) N = 25

Elegir valores para la resolución y el radio que convengan.

Como una extensión al algoritmo, podemos agregar un ángulo inicial, $ángulo_i$. Esto nos ayudará para "colocar" el primer vértice donde queramos. Consecuentemente, los demás vértices también serán colocados según el ángulo inicial. Esto implica que la figura será rotada tantos radianes desde la horizontal, que pasa por el centro geométrico del polígono al igual que el de la circunferencia. El algoritmo extendido es el siguiente:

```
Polígono_Regular( N, centro, radio, ángulo_i )
1.  Crear una lista de N vértices: vértices
2.  alfa <- 2π / N
3.  Bucle: i <- 1 hasta N
4.     vértices[i].x <- centro.x + radio * cos( (i-1)*alfa + ángulo_i )
5.     vértices[i].y <- centro.y + radio * sen( (i-1)*alfa + ángulo_i )
```

```
6.  Dibujar_Polígono( vértices, N )
7.  Terminar.
```

2. Escribir un programa para dibujar una estrella. Para esto, seguimos el algoritmo para dibujar un polígono regular. Sin embargo, a la hora de dibujar el polígono, necesitamos implementar nuestra propia función que va dibujando cada línea de un vértice a otro vecino, pero saltándose cada 2,3,4,...,N/2 vértices en nuestra lista. El algoritmo es simplemente el siguiente:

```
Estrella( N, centro, radio, ángulo_i, salto )
1.  Crear una lista de N vértices: vértices
2.  alfa <- 2π / N
3.  Bucle: i <- 1 hasta N
4.     vértices[i].x <- centro.x + radio * cos( (i-1)*alfa + ángulo_i )
5.     vértices[i].y <- centro.y + radio * sen( (i-1)*alfa + ángulo_i )
6.  Dibujar_Estrella( vértices, N, salto )
7.  Terminar.
```

El algoritmo de *Dibujar_Estrella()* se basa en manipular los índices de nuestra lista de vértices para así dibujar líneas entre un vértice y el siguiente según el valor de *salto*. Por lo tanto, nos basaremos en la operación del módulo o resto de una división, descrito como **mod**. He aquí el algoritmo:

```
Dibujar_Estrella( vértices, N, salto )
1.  Bucle: i <- 0 hasta N-1
2.     j <- (i+salto) mod N
3.     Dibujar_Línea( vértices[i+1].x, vértices[i+1].y, vértices[j+1].x, vértices[j+1].y )
4.  Terminar.
```

Probar con los siguientes valores:
a) N = 7, salto = 2
b) N = 7, salto = 3
c) N = 10, salto = 4
d) N = 12, salto = 5
e) N = 25, salto = 5
f) N = 25, salto = 12

3. Mostrar las estrellas anteriores, pero sólo los perímetros. Es decir, sin que se vean las líneas interiores de la estrella, en sí. Para esto, tenemos que calcular los vértices donde se cruzan las líneas al crear la estrella. Las líneas cruzantes indican un punto común para ambas líneas. Esto implica que tenemos que resolver el siguiente sistema de ecuaciones, para averiguar las coordenadas *x* e *y* de tal punto común:

$$y = (B_y - A_y) / (B_x - A_x) * (x - A_x) + A_y,$$
$$y = (D_y - C_y) / (D_x - C_x) * (x - C_x) + C_y$$

donde tenemos las líneas, o mejor dicho los segmentos, AB y CD.

Simplificando, obtenemos que,

$$x = (m*A_x - A_y - n*C_x + C_y) / (m-n),$$
$$y = m*(x - A_x) + A_y,$$
donde,
$$m = (B_y - A_y) / (B_x - A_x), \text{ y}$$
$$n = (D_y - C_y) / (D_x - C_x)$$

Algunas sugerencias para la implementación:
1. Calcula los vértices de la estrella, como se ha explicado anteriormente, en el ejercicio #2. Luego, calcula los vértices como se ha explicado previamente, en este ejercicio. Esto implica que nuestra lista de vértices ahora será de 2*N, ya que existe el doble de vértices que en el caso anterior de la estrella. El segmento AB se basa en el vértice ***i*** y el siguiente, ***i+salto***, mientras que el segmento CD se crea con el vértice contiguo ***i+1*** y su anterior, ***i+1-salto***. Por supuesto, hay que tener en cuenta que no podemos sobrepasar los valores de los índices. Esto implica que necesitamos "dar la vuelta", por lo que necesitaremos usar la operación del módulo.

2. En el cálculo de las coordenadas del punto común *(x,y)*, debemos asegurarnos de que ***m*** y ***n*** sean valores determinables (calculables). Si nos fijamos, cada valor se basa en una división, por lo que es posible que el denominador sea cero. Consecuentemente, ***m*** o ***n*** no podrían ser calculados. Si esto ocurre, entonces implica que uno de los segmentos es vertical: AB o CD. En este caso, sabemos el valor de ***x***, ya que éste no varía, al ser un segmento vertical; o sea, $x = A_x = B_x$ o $x = C_x = D_x$. Sabiendo esto, sólo tenemos que calcular el valor de ***y***, a partir del sistema de ecuaciones presentado anteriormente. Se nos presenta un caso similar, si el segmento es horizontal, que ocurriría cuando ***m*** o ***n*** es cero - cuando el numerador es cero. En este caso, conoceremos el valor de ***y***, por lo que podemos calcular fácilmente el valor de ***x***. Hay que tener en cuenta que surge un problema cuando el denominador de tanto ***m*** o ***n*** es cero, pero no existe ningún problema si el numerador de ***m*** o ***n*** es cero, únicamente que podemos simplificar los cálculos.

4. Crear un programa para mostrar estrellas, como se ha explicado en el ejercicio #3. Esta vez, modificaremos el radio según el valor del ángulo basado en ***alfa***. Es decir, el radio será ahora una función dependiente de una expresión conteniendo ***alfa***. El algoritmo será parecido al siguiente:

```
4.    vértices[i].x <- centro.x + radio( (i-1)*alfa + ángulo_i ) * cos( (i-1)*alfa + ángulo_i )
5.    vértices[i].y <- centro.y + radio( (i-1)*alfa + ángulo_i ) * sen( (i-1)*alfa + ángulo_i )
```

Probar con los siguientes valores y funciones:
a) N = 25, salto = 11,

```
Real radio( t )
1.  a <- 12
2.  b <- 30
3.  x <- a*cos( t ) + b*sen( a*t/2 )
4.  y <- a*cos( t ) - b*sen( a*t/2 )
5.  resultado <- √( (x²+y²) / ((a+b)² + (a-b)²) )
6.  Terminar( resultado )
```

b) N = 12, salto = 5,

```
Real radio( t )
1.  a <- 12
2.  b <- 30
3.  x <- a*cos( t ) + b*sen( a*t/2 )
4.  y <- a*cos( t ) - b*sen( a*t/2 )
5.  resultado <- √( (x²+y²) / ((a+b)² + (a-b)²) )
6.  Terminar( resultado )
```

5. Basándose en la figura 17, dibujar otras dos figuras:
a) Extendiendo la cara agregando un cuerpo con brazos y piernas, y
b) Diseñando otra cara expresando otro sentimiento, como por ejemplo: sorpresa, sueño, enfado, perplejidad, etc..

6. Usando la técnica de composición, dibujar una casa. Pueden basarse en la siguiente ilustración de la figura 24.

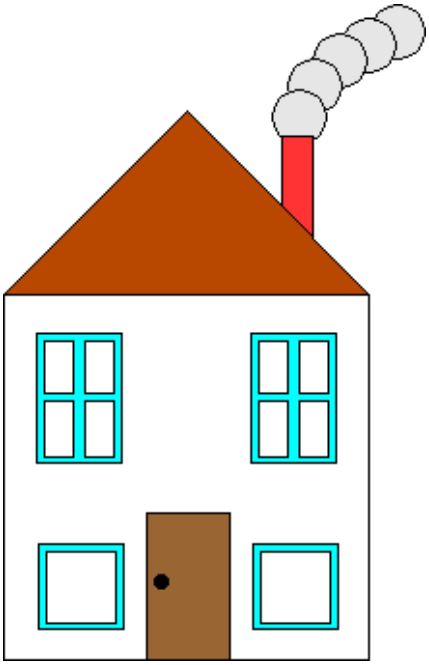


Figura 24 - Casa

7. Crear un programa que recoja datos de un fichero y mostrarlos en dos gráficas en forma de a) barras y b) círculo, junto con algunos resultados estadísticos, si se desea. Por ejemplo, mostrar los porcentajes de cada sector del gráfico circular incluyendo una leyenda para indicar la representación de cada color. También se podría calcular e indicar visualmente el promedio, varianza, desviación estándar, etc. en la gráfica de barras.

8. Crear un programa que permita construir un diagrama de un circuito eléctrico según los datos guardados en un fichero o indicado directamente por el usuario. Por ejemplo, el programa podría crear un circuito con dos generadores eléctricos, cuatro condensadores, tres resistencias, dos inductores, y un interruptor. La complejidad del programa es responsabilidad del programador; se puede crear un programa altamente adaptable a cada tipo de circuito o un programa muy simple que agrega un componente detrás de otro, secuencialmente.

Como una sugerencia, crean funciones para poder dibujar cada componente de forma general. El programa principal invocará cada función para dibujar el componente eléctrico según los datos que describen el circuito. La descripción del circuito puede ser simple, como una cadena de caracteres. Por ejemplo, "GRRCCILGLCCR", donde G indica un generador, R es una resistencia, C es condensador, I es interruptor, y L es inductor.

9. Diseñar un campo de fútbol. Se puede basar en el diagrama de la figura 25.

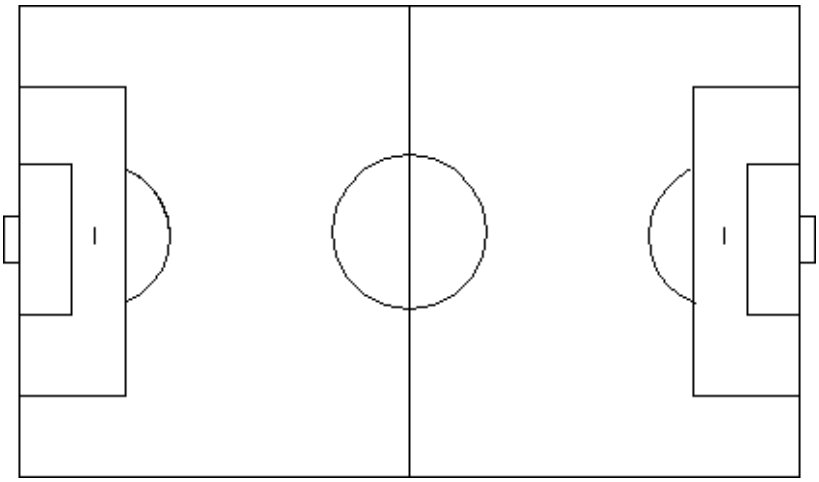


Figura 25 - Campo de fútbol

10. Diseñar una figura usando curvas de Bézier. No es necesario que el dibujo se base exclusivamente en curvas de Bézier. Por ejemplo, intentar diseñar un logotipo propio o quizá copiarse de uno existente. Otro ejemplo puede ser recrear una letra del abecedario usar estas curvas. Se aconseja hacer uso del applet de Java presentado anteriormente en este capítulo para diseñar curvas de Bézier.

Descarga de Paquetes

En esta página ofreceremos varias versiones de paquetes que están ligadas a un capítulo específico de nuestro curso de gráficos. Cada paquete es un fichero en formato ZIP que contiene los archivos necesarios, pero básicos, para poder realizar los ejercicios y ejemplos de cada capítulo del curso. Los archivos son 2 ficheros fuentes: *principalXX.c* y *dibujarXX.c* y 1 fichero de cabecera: *dibujarXX.h*, donde *XX* se refiere al capítulo de forma numérica. Estos archivos contienen suficiente código escrito para poder crear una aplicación para MS-Windows®. Algunos ejercicios pedirán que se implemente una función según el tema tratado del capítulo, por lo que esos paquetes particulares no contendrán una función equivalente. En posteriores versiones, los paquetes posiblemente contendrán la función en cuestión, usando el API de MS-Windows®, si existe tal funcionalidad. Estos paquetes no son esenciales para seguir el curso ni para realizar los ejercicios ni ejemplos. La razón de incluir tales paquetes es para ayudar a aquellas personas que no sepan usar el API de MS-Windows®, ni librerías o API's gráficas. Tales y como están diseñados los ficheros de cada paquete, el alumno o la alumna deberá alterar la función *Dibujar()* dentro del fichero *dibujarXX.c*; existen comentarios para guiar al/a la programador/a. Estos paquetes sirven a modo de "plantilla", por lo que los alumnos y alumnas pueden alterar su funcionalidad como les convenga.

Capítulo 2

Las nuevas funciones, macro, y variables contenidas son:

Prototipo	**Descripción**
void Dibujar(void)	Dibuja la escena total. La única función, contenida en este paquete, que debe ser implementada por el/la estudiante.
void Linea(int x_comienzo, int y_comienzo, int x_fin, int y_fin)	Traza una línea recta desde (**x_comienzo**,**y_comienzo**) hasta (**x_fin**,**y_fin**), usando el color seleccionado. Las dimensiones de las coordenadas están en píxeles.
Macro	**Descripción**

CambiarColorPincel(int r, int g, int b)	Cambia el color seleccionado para el pincel en uso. El pincel sirve para dibujar líneas rectas y píxeles individuales. Los parámetros **r**, **g**, y **b** representan los componentes básicos: rojo, verde, y azul, los cuales pueden tener, cada uno, un valor entre [0, 255], que describen la intensidad de cada componente.
Variable Global	**Descripción**
extern HDC hImagen	Contexto gráfico de la ventana. La escena es dibujada en este contexto.
extern const UINT uiAltura	La altura (total) de la ventana en píxeles. El/la estudiante puede alterar este valor.
extern const UINT uiAnchura	La anchura (total) de la ventana en píxeles. El/la estudiante puede alterar este valor.

Descargar el paquete **capitulo02.zip**.

Capítulo 3

Las nuevas macros y funciones contenidas son:

Macro	**Descripción**
COLOR	Tipo COLORREF que guarda la información de un color en un dato de 32 bits. Este tipo pertenece al API de MS-Windows®.

CambiarColorBrocha(int r, int g, int b)	Cambia el color seleccionado para la brocha en uso. La brocha sirve para rellenar polígonos. Los parámetros **r**, **g**, y **b** representan los componentes básicos: rojo, verde, y azul, los cuales pueden tener, cada uno, un valor entre [0,255], que describen la intensidad de cada componente.
PonPixel(int x, int y, COLOR color)	Activa un solo píxel en la posición (**x**,**y**) de un color de tipo COLOR.
PUNTO2	Estructura POINT que contiene dos campos: **x** e **y** de tipo LONG. Esta estructura pertenece al API de MS-Windows®.
RGB(int r, int g, int b)	Crea un valor de tipo COLOR. La macro RGB() pertenece al API de MS-Windows®.
TraePixel(int x, int y)	Averigua el color de tipo COLOR de un solo píxel en la posición (**x**,**y**).
Prototipo	**Descripción**
void TrianguloRelleno(PUNTO2 A, PUNTO2 B, PUNTO2 C)	Dibuja un triángulo con el pincel seleccionado y rellena su interior con la brocha seleccionada. La función acepta 3 argumentos que contienen las coordenadas en píxeles.
void RectanguloRelleno(PUNTO2 A, PUNTO2 B)	Dibuja un cuadrado con el pincel seleccionado y rellena su interior con la brocha seleccionada. La función acepta 2 argumentos que contienen las coordenadas en píxeles de la esquina superior izquierda, **A** y de la esquina inferior derecha, **B** que forman parte del rectángulo.

void PoligonoRelleno(PUNTO2 *pListaPuntos, int nCantidadPuntos)	Dibuja un polígono con el pincel seleccionado y rellena su interior con la brocha seleccionada. La función acepta 2 argumentos: **pListaPuntos** es una lista de puntos, en píxeles, representando las vértices del polígono, y **nCantidadPuntos** indica la cantidad de vértices de tal lista. La función cerrará el polígono, dibujando una línea del último vértice al primero.

Descargar el paquete **capitulo03.zip**.

Capítulo 4

No existen nuevas funciones ni macros.

Descargar el paquete **capitulo04.zip**.

Capítulo 5

No existen nuevas funciones ni macros.

Descargar el paquete **capitulo04.zip**.