Curso de gráficos

Gráficos en programación

Steven R. Davidson

http://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 dinámicamente enlazadas: árboles, colas, pilas, etc..
- Matemáticas pre-universitarias: ecuaciones/funciones, geometría, trigonometría, vectores, etc..

Es preferible que el lector tenga conocimientos de alguna biblioteca 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 biblioteca 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:

Mosaico

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, RGBA, 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 el suavizado de imagen (o "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 René Magritte, *Ceci n'est pas une pipe* (1928).

"Esto no es una pipa"

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 800 x 600.

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 mínimo y máximo a usarse en la ecuación. Digamos que queremos ver parte de la gráfica usando los valores de $\boldsymbol{x}$: **[-3, +3]**; o sea, $\boldsymbol{x_{ui} = -3}$ y $\boldsymbol{x_{uf} = 3}$, los cuales representan los valores mínimo y máximo 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: $\boldsymbol{x_{ui} = -3 \Rightarrow 0}$ y $\boldsymbol{x_{uf} = 3 \Rightarrow 799}$. 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, $\boldsymbol{x_p}$ = 345, para poder averiguar el valor de $\boldsymbol{x}$ 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, $\boldsymbol{x_u}$ = -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 $\boldsymbol{y}$ 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 $y_u = 3\ (-2{,}0000)^2 - 6 \Rightarrow y_u = 6{,}0000$. 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: $\boldsymbol{y = mx + b}$, $\boldsymbol{m}$: [0,1], donde $\boldsymbol{m}$ 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: $\boldsymbol{x_{uf} = 3}$ e $\boldsymbol{y_{uf} = 21}$; sus homólogos en píxeles son: $\boldsymbol{x_{pf} = 800}$ e $\boldsymbol{y_{pf} = 600}$. 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 permitirá 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*.

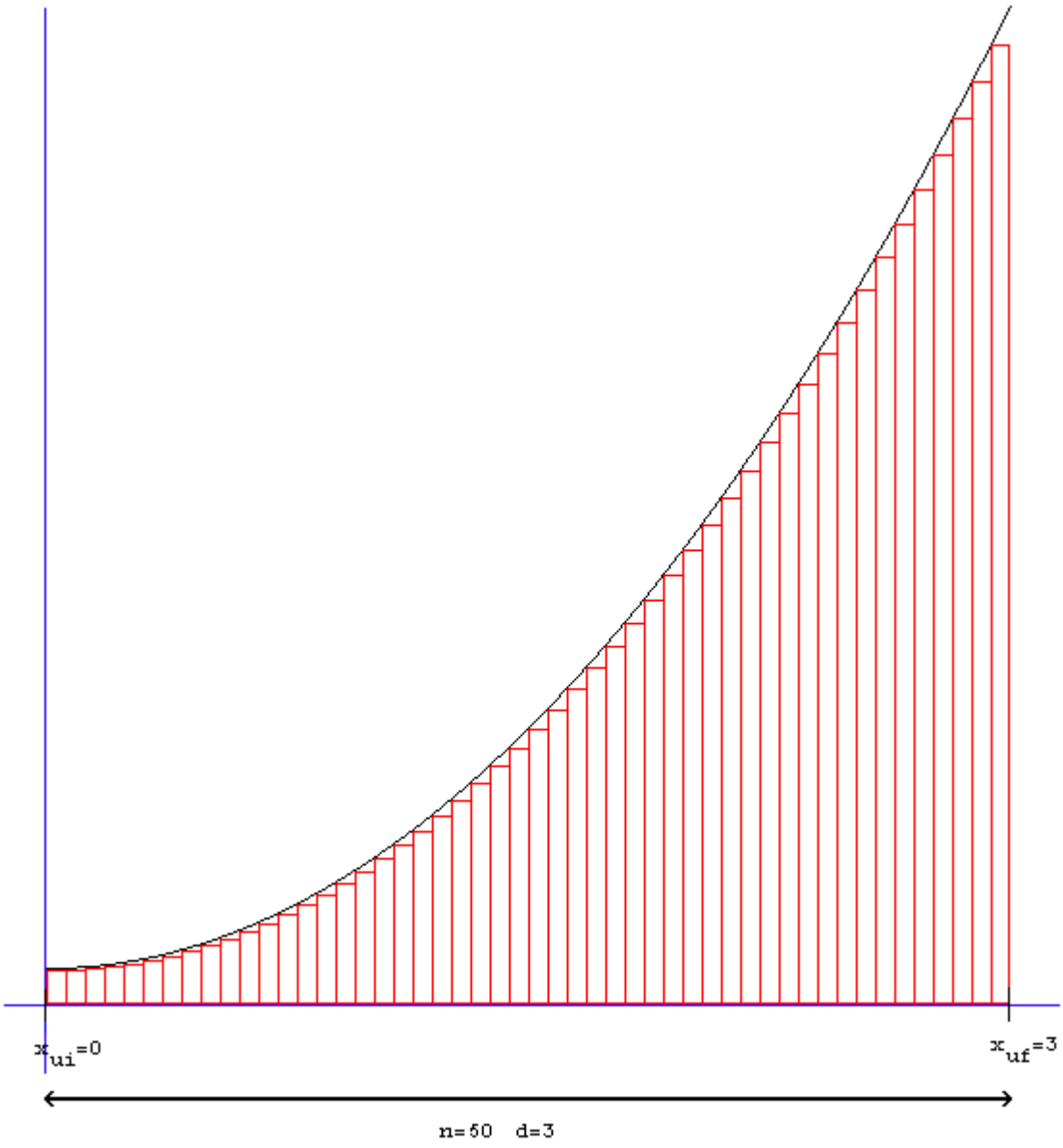


Figura 1

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

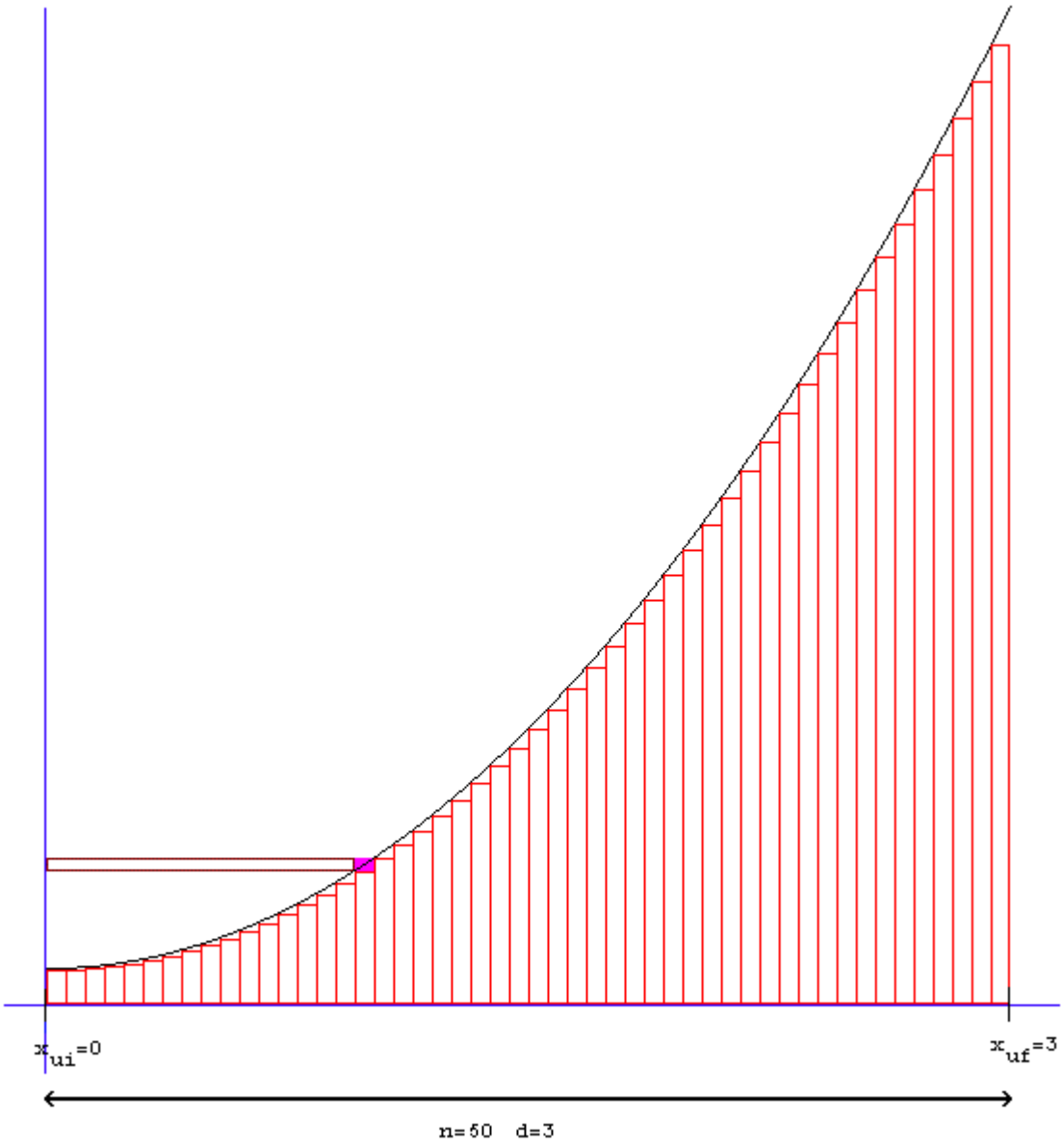


Figura 2

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:

$x_0 = [x_{ui}, x_{ui} + 3/50)$
$x_1 = [x_{ui} + 3/50, x_{ui} + 6/50)$
$x_2 = [x_{ui} + 6/50, x_{ui} + 9/50)$
...
$x_a = [x_{ui} + a*d / n, x_{ui} + (a+1)*d / n), \quad a = 0,1,2,...,n-1$

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:

$$\frac{600 \text{ píxeles}}{\text{------------}} = \frac{|(y_{pi} + \text{altura}_p) - y_p| \text{ píxeles}}{\text{------------------------------}} \Rightarrow$$

$$\frac{27 \text{ unidades}}{(6{,}0000 - y_{ui}) \text{ unidades}}$$

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

$y_p = 333{,}3333$ píxeles $\Rightarrow y_p = 333$ 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:

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

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

$y_p = 333{,}3333$ píxeles $\Rightarrow y_p = 333$ 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
$anchura_p$ $\lvert x_{pf} - x_{pi} + 1 \rvert$	Número de columnas a representar en píxeles
$altura_p$	Número de filas a

$\lvert y_{pf} - y_{pi} + 1 \rvert$	representar en píxeles
$anchura_u$ $\lvert x_{uf} - x_{ui} \rvert$	Distancia entre los valores final e inicial de las coordenadas X en unidades cartesianas
$altura_u$ $\lvert y_{uf} - y_{ui} \rvert$	Distancia entre los valores final e inicial de las coordenadas Y en unidades cartesianas

Fórmulas Usadas

$\frac{anchura_u}{anchura_p} = \frac{\lvert x_u - x_{ui} \rvert}{\lvert x_p - x_{pi} \rvert}$	Para hallar el valor de x_u conociendo el valor de x_p
$\frac{altura_p}{altura_u} = \frac{\lvert (y_{pf} - y_{pi} + 1) - y_p \rvert}{\lvert y_u - y_{ui} \rvert}$	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_{uf}, x_{pi}, x_{pf}, y_{pi}, y_{pf}
2. $anchura_u \leftarrow |x_{uf} - x_{ui}|$
3. $altura_u \leftarrow |y_{uf} - y_{ui}|$
4. $anchura_p \leftarrow |x_{pf} - x_{pi} + 1|$
5. $altura_p \leftarrow |y_{pf} - y_{pi} + 1|$
6. $dx_{up} \leftarrow anchura_u / anchura_p$
7. $dy_{pu} \leftarrow altura_p / altura_u$
8. $x_u \leftarrow x_{ui}$
9. $y_u \leftarrow$ función(x_u)
10. $x\_orig_p \leftarrow x_{pi}$
11. $y\_orig_p \leftarrow y_{pf} + 1 - dy_{up} * |y_u - y_{ui}|$
12. Bucle: $x_p \leftarrow x_{pi}+1$ hasta x_{pf} con incremento de 1
    13. $x_u \leftarrow x_u + dx_{up}$
    14. $y_u \leftarrow$ función(x_u)
    15. $y_p \leftarrow y_{pf} + 1 - dy_{up} * |y_u - y_{ui}|$
    16. Dibujar_Línea(x_orig_p, y_orig_p, x_p, y_p)
    17. $x\_orig_p \leftarrow x_p$
    18. $y\_orig_p \leftarrow y_p$

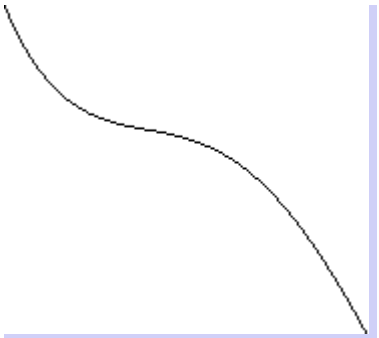
función() es la ecuación cuya gráfica queremos trazar. La función *Dibujar_Línea()* hace alusión a la función básica de cualquier API o biblioteca gráfica, para trazar una línea recta desde una coordenada de píxeles a otra.

Ejercicios

Enlace al paquete *de este capítulo.*

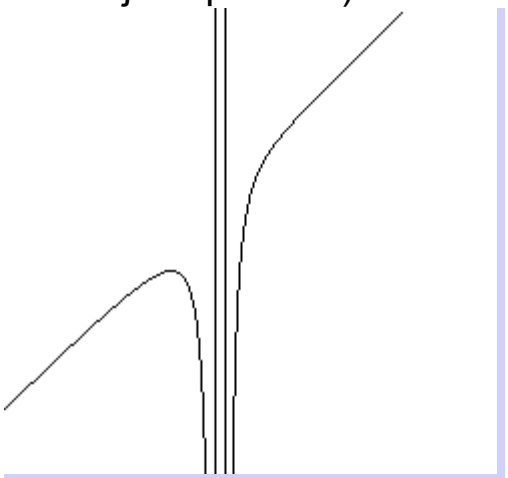
1. Escriba, para cada ecuación, un programa que trace las gráficas de las siguientes ecuaciones y según las restricciones dadas:
   a. $y(x) = 0,5x^4 - 2x^3 + 0,33x^2 - 0,75x + 1$,
      Restricciones: $x = [-1, +2]$, $y = [y(2), y(-1)]$,

Resolución: 100 x 100.



Ejercicio 1a

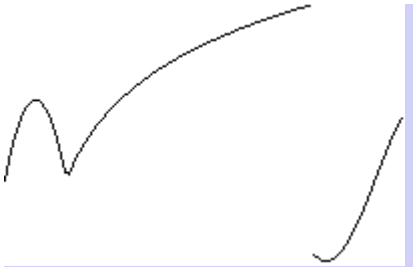
b. $y(x) = (1,5x^3 - 2,6x^2 + 3,3x - 10) / (x^2 - 1,5x + 0,5)$,
Restricciones: $x = [0, +2]$, y = elija un intervalo apropiado,
Resolución: 300 x 300,
Observación: la ecuación contiene dos asíntotas verticales, x_1 y x_2. Calcula, con anterioridad, estos valores de x para que la ecuación no sea indeterminada en $x = x_1$ ni en $x = x_2$.
(Pista: Precalcula, $x^2 - 1,5x + 0,5 = 0$, 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).



Ejercicio 1b

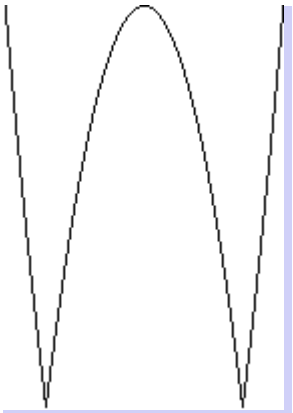
c. Si $x < 1$, entonces $y(x) = 1 - x^2$, si $1 \leq x \leq 9$, entonces $y(x) = \ln x$, si $x > 9$, entonces $y(x) = \cos x$,
Restricciones: $x = [-1, +12]$, y = elija un intervalo

apropiado,
Resolución: 600 x 600.



Ejercicio 1c

d. `y(x) = |6 - 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.



Ejercicio 1d

2. Elija dos de las ecuaciones anteriores y reescriba sus programas para que trace las curvas usando el método descrito en el último párrafo de la sección *Observaciones*, bajo b)2.. Esto es, trace la ecuación a partir de valores dados para y_p en vez de x_p. Compare las gráficas obtenidas con este método junto con el usado en el ejercicio anterior.
3. Escriba un programa para trazar la siguiente ecuación:
`f(x) = 4 / π * [ ∑( sen(nx) / n ) ]`, donde

n = 1,3,5,7,..., +∞ (infinito).

Si escribimos unos cuantos términos de la suma, obtenemos lo siguiente:

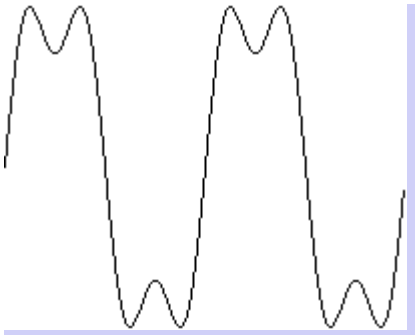
f(x) = 4 / π * [sen(x) + sen(3x) / 3 + sen(5x) / 5 + sen(7x) / 7 + sen(9x) / 9 + sen(11x) / 11 + sen(13x) / 13 + ...],

Restricciones: x = [-2π, +2π], y = [-1,5, +1,5],

Resolución: 800 x 800,

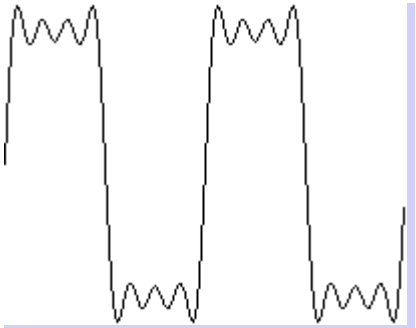
Como no podemos llegar hasta +*infinito*, entonces elegiremos un intervalo *finito*. Esto es, *n* := [1,N]. Para el valor de N, use:

a. N = 3,



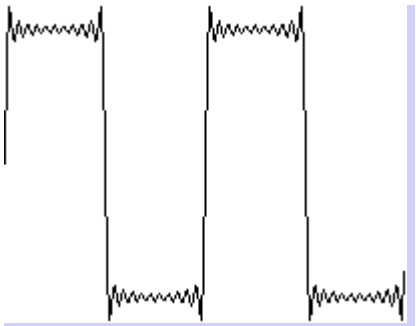
Ejercicio 3a

b. N = 7,



Ejercicio 3b

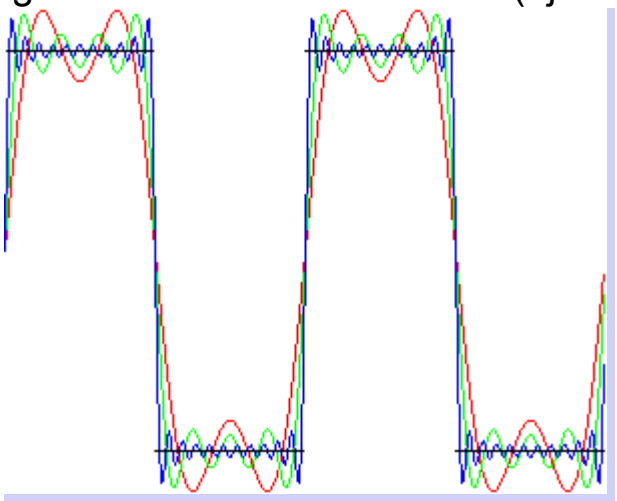
c. N = 21,



Ejercicio 3c

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 a un valor insignificante.

4. La ecuación presentada en el ejercicio #3 es, en realidad, la serie de Fourier para esta ecuación periódica:
   Si $-\pi < x < 0$, entonces $f(x) = -1$. Si $0 < x < +\infty$, entonces $f(x) = 1$.
   Reescriba el programa del ejercicio anterior #3 - usando las mismas restricciones y resolución - para dibujar la función anterior, presentada en este ejercicio #4. Si sabe manipular colores, use distintos colores para cada trazado. Si no puede distinguir un trazado del otro, entonces cree otro programa diferente y ejecute ambos simultáneamente. Compare 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).

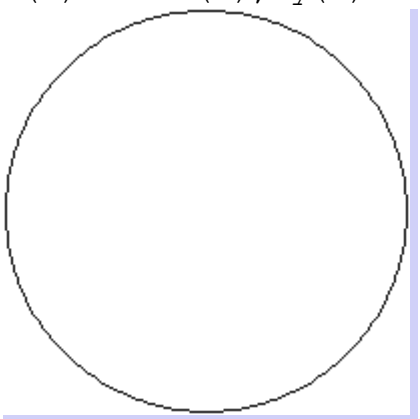


Ejercicio 4

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 $x_1 = -\pi$, $x_2 = 0$, y $x_3 = +\pi$ no están definidos para `f(x)`.

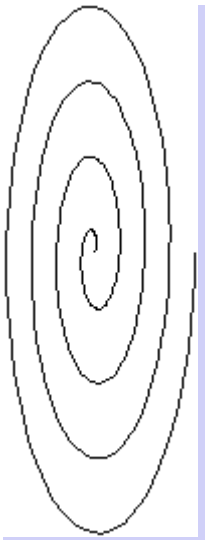
2) La ecuación anterior debe ser trazada periódicamente. Es decir, el mismo trazado debe ser igual para los intervalos `x := ... ∪ (-3π, -2π) ∪ (-2π, -π) ∪ (-π, 0) ∪ (0, +π) ∪ (+π, 2π) ∪ (+2π, +3π) ∪ ...`. La periodicidad es de 2π; o sea, cada 2π unidades, la gráfica se repite.

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. Escriba un programa para trazar las siguientes curvas paramétricas:
   a. `x(t) = sen(t), y(t)= cos(t)`, donde 0 ≤ t ≤ 2π,



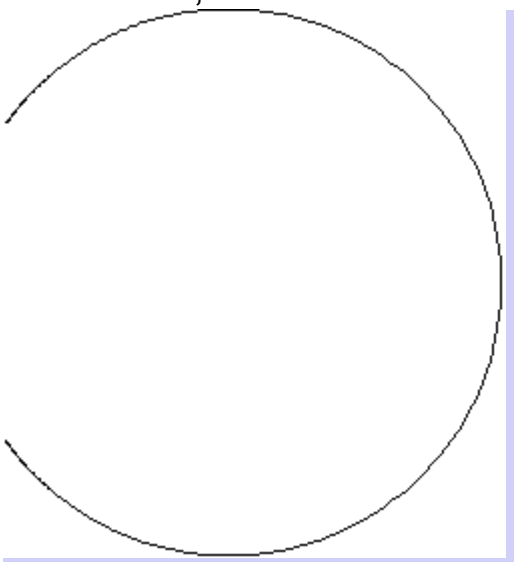
Ejercicio 5a

   b. `x(t) = t*cos(t), y(t) = 3t*sen(t)`, donde 0 ≤ t ≤ 8π,



Ejercicio 5b

c. `x(t) = (1 - t^2) / (1 + t^2), y(t) = 2t / (1 + t^2)`, donde $-\pi \leq t \leq +\pi$,

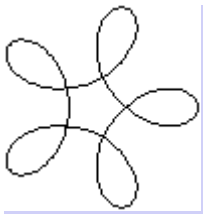


Ejercicio 5c

d. `x(t) = a*cos(t) + b*cos(at/2),`
`y(t) = a*sen(t) - b*sen(at/2)`, donde $0 \leq t \leq 2\pi$,

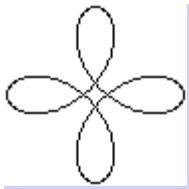
Esta curva se denomina hipotrocoide. Pruebe con unos cuantos valores para *a* y *b* y con diferentes intervalos de *t*; por ejemplo,

1. a = 8, b = 5



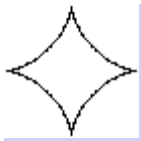
Ejercicio 5d1

2. a = 6, b = 5



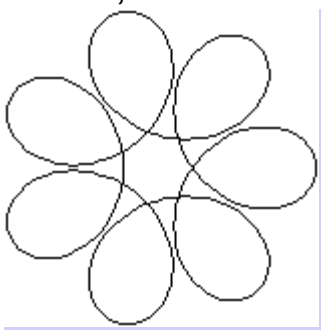
Ejercicio 5d2

3. a = 6, b = 2



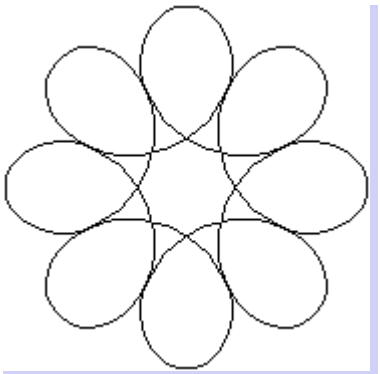
Ejercicio 5d3

4. a = 12, b = 8



Ejercicio 5d4

5. a = 14, b = 8,6



Ejercicio 5d5

Elija 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 es (*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. Los incrementos de los valores de *t* no son fácilmente visible. Intente 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 - Koch

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*.

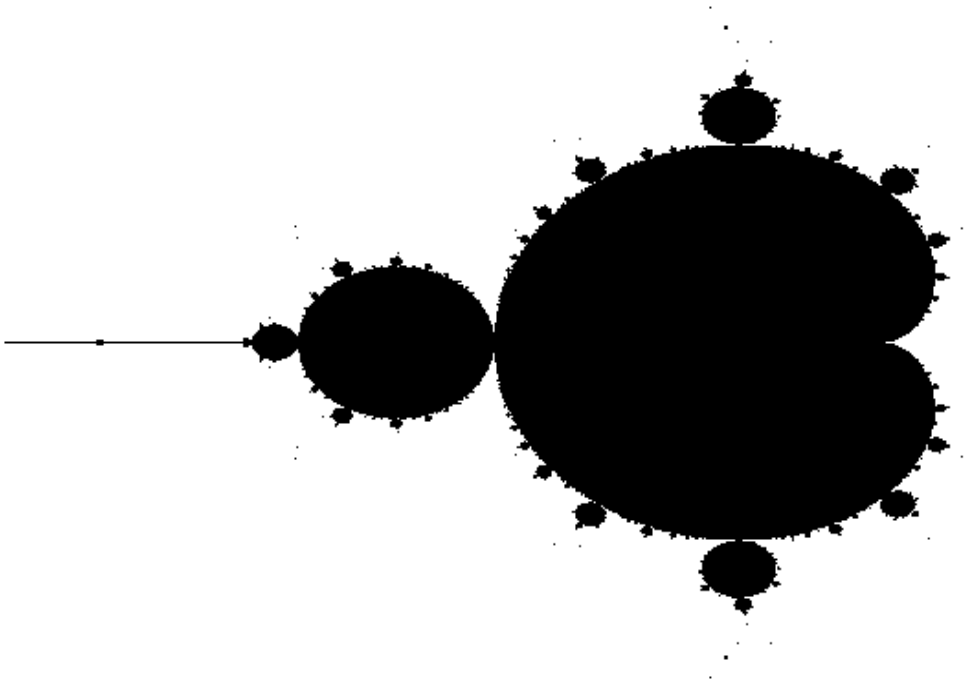


Figura 2 - Mandelbrot

Explicaremos estos fractales en mayor detalle a continuación.

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 - Koch n=0

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 - Koch n=1

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*.

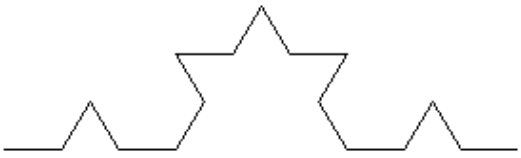


Figura 5 - Koch n=2

Podríamos reescribir la definición de esta regla para reflejar la recursividad:

$A_0 \rightarrow$ Avanzar

$A_{n+1} \rightarrow A_n I A_n D D A_n I A_n$

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.ángulo ← 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.ángulo ← Cursor.ángulo + π/3
            // I
         7. Dibujar_Koch( Cursor, Dist/3, N-1 )
            // A
         8. Cursor.ángulo ← Cursor.ángulo - π*2/3
            // DD
         9. Dibujar_Koch( Cursor, Dist/3, N-1 )
            // A
        10. Cursor.ángulo ← Cursor.ángulo + 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.ángulo )
 3. P.y ← Cursor.punto.y + D*sen( Cursor.ángulo )
 4. Dibujar_Línea( Cursor.punto.x, Cursor.punto.y, P.x, P.y )
 5. Terminar( P )
```

D es la distancia o longitud del segmento a trazar.
Dibujar_Línea() 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.ángulo*.

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_Línea()* 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:

$$d_{cap} = - \lim_{n\to\infty} \frac{\ln N_n}{\ln L_n} = - \lim_{n\to\infty} \frac{\ln 4^n}{\ln 3^{-n}} = \lim_{n\to\infty} \frac{n * \ln 4}{-n * \ln 3} = \lim_{n\to\infty} \frac{\ln 4}{\ln 3} = \frac{\ln 4}{\ln 3} = 1{,}261859507...$$

(Nota: ∞ 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:

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

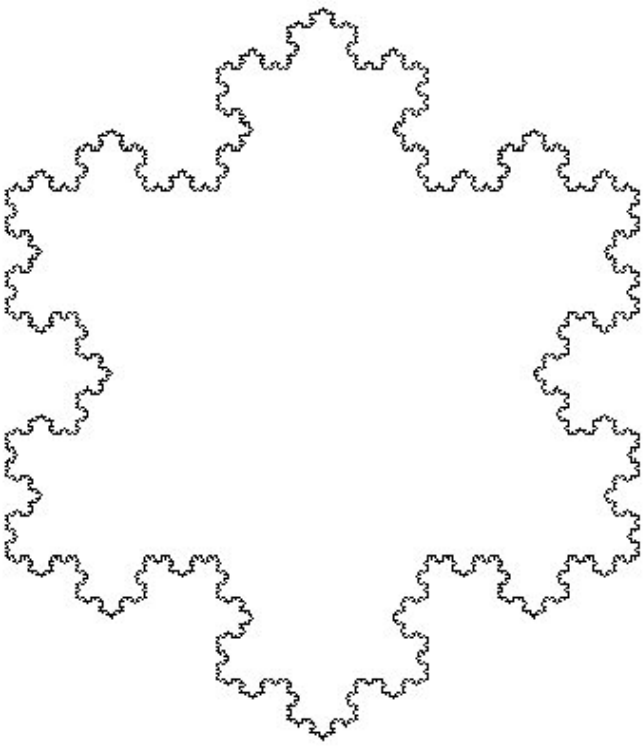
Aquí vemos que $\mathbf{d_{cap}}$ es el exponente para $\mathbf{L_n}$ que equivale a $\mathbf{N_n}$.

Ejercicios

Enlace al paquete *de este capítulo.*

1. Escriba un programa que dibuje la curva de Koch. Se puede implementar el algoritmo mostrado en el capítulo. Prueben 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. Prueben con una resolución aceptable: 300x300 ó 500x500. También se puede cambiar el ángulo inicial de 0 radianes a π/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° (=π/3 radianes) para formar el triángulo equilátero a partir de la "base".
   A_0 → Avanzar,
   A_{n+1} → $A_nIA_nDDA_nIA_n$,

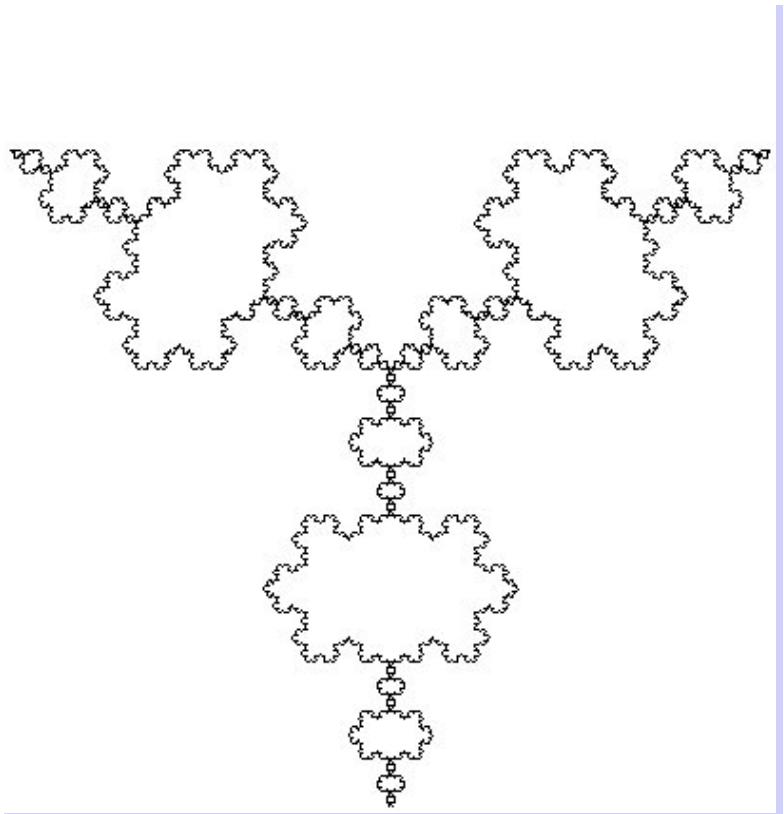
   Hemos agregado otra "regla", B, para describir la estructura que formará la base: un triángulo equilátero. Básicamente, estaremos dibujando 3 curvas de Koch situadas en cada lado del triángulo, con las "puntas" hacia fuera. Después de unas cuantas iteraciones, la figura dará forma a un copo de nieve.



Copo de Nieve - N=5

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

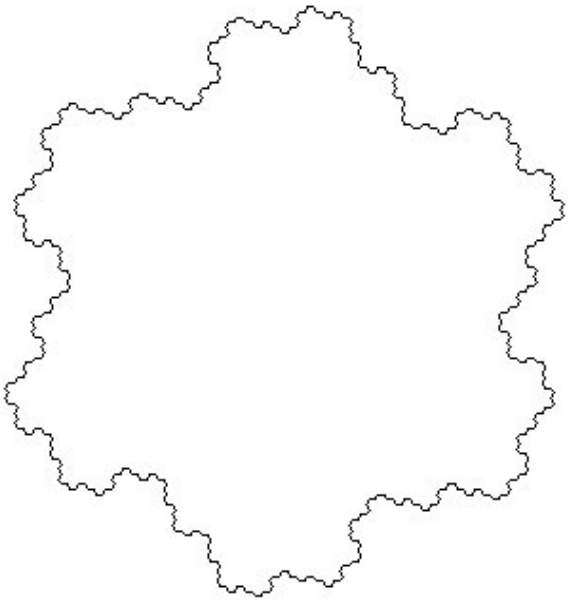
3. Otro ejemplo popular es realizar el Anti-Copo de Nieve de Koch. Esto consiste en dibujar las curvas de Koch con las puntas hacia el interior del triángulo al igual que tener un triángulo invertido; el triángulo se apoya con su pico y con un lado hacia en la parte superior en vez de en la inferior. 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,
      $A_{n+1} \rightarrow A_n D A_n I I A_n D A_n$, o
   b. B para que la forma de dibujar el triángulo equilátero básico ya tenga una orientación inversa. Esto implicaría que,
      $B \rightarrow I A_n I I A_n I I A_n$, a partir del "pico" del triángulo invertido.



Anti-Copo de Nieve - N=5

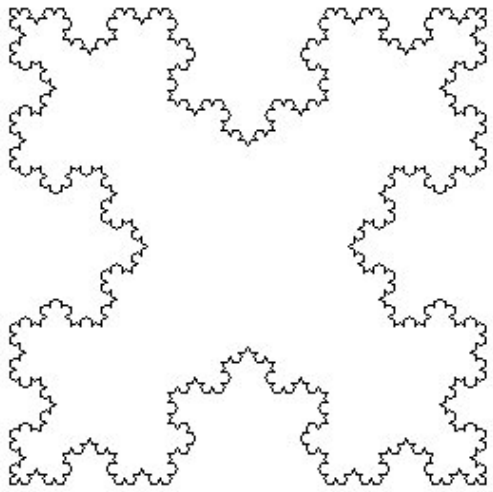
Escriba un programa que dibuje el anti-copo de nieve descrito en este ejercicio.

4. Con estas reglas de producción, podemos construir muchos tipos de figuras. Escriba un programa para dibujar cada figura descrita por las siguientes reglas de producción:
   a. La Isla de Gosper,
      $B \rightarrow A_nDA_nDA_nDA_nDA_nDA_n$, describe un hexágono regular,
      $A_0 \rightarrow$ Avanzar,
      $A_{n+1} \rightarrow A_nIA_nDA_n$,
      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.
      Calcule A_4.



Isla de Gosper - N = 4

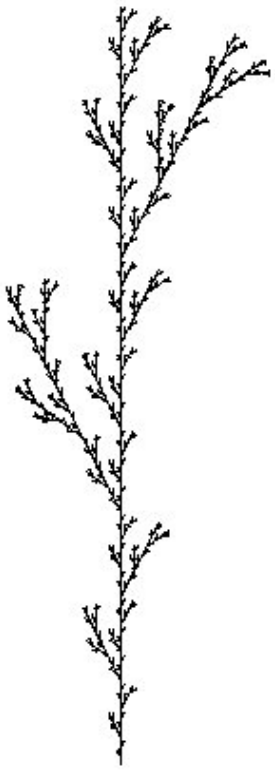
b. Fractal de Cesàro,
   B → A_nIA_nIA_nIA_n, con un ángulo de 90° (=π/2 radianes), describe un cuadrado,
   A_0 → Avanzar,
   A_{n+1} → A_nIA_nDDA_nIA_n,
   Los giros de A_n se hacen con un ángulo de 60° (=π/3 radianes). Esta regla es la misma que la curva de Koch, pero la regla de la base es un cuadrado. 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.
   Calcule 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. Observen que la información guardada es tanto la posición como el ángulo del *Cursor*.
   Escriba un programa que dibuje una figura para cada una de las siguientes reglas de producción:
   a. árbol sin hojas pero con muchas ramas,
      $A_0 \rightarrow$ Avanzar,
      $A_{n+1} \rightarrow A_n[IA_n]A_n[DA_n]A_n$,
      ángulo inicial: 90° (=π/2 radianes) - para que el árbol esté "de pie".

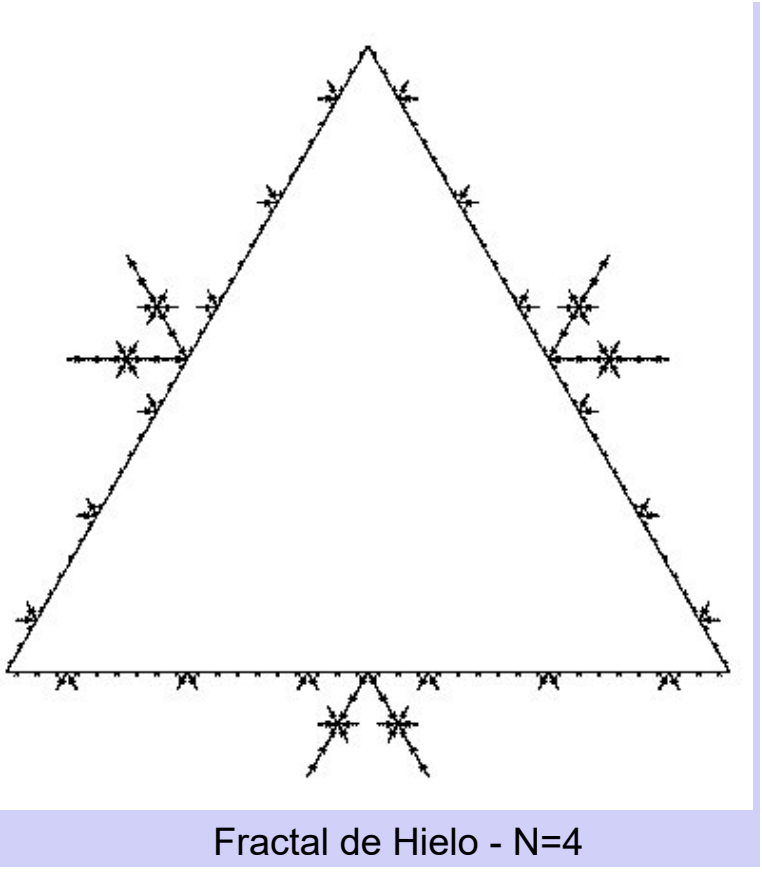
Todos los giros se hacen con un ángulo de 27° (= 0,471239 radianes).
Resolución: 400x400.
Longitud original (del Avance): 400 píxeles.
División del tramo: d=1/3.
Calcule A_6.



Árbol - N=6

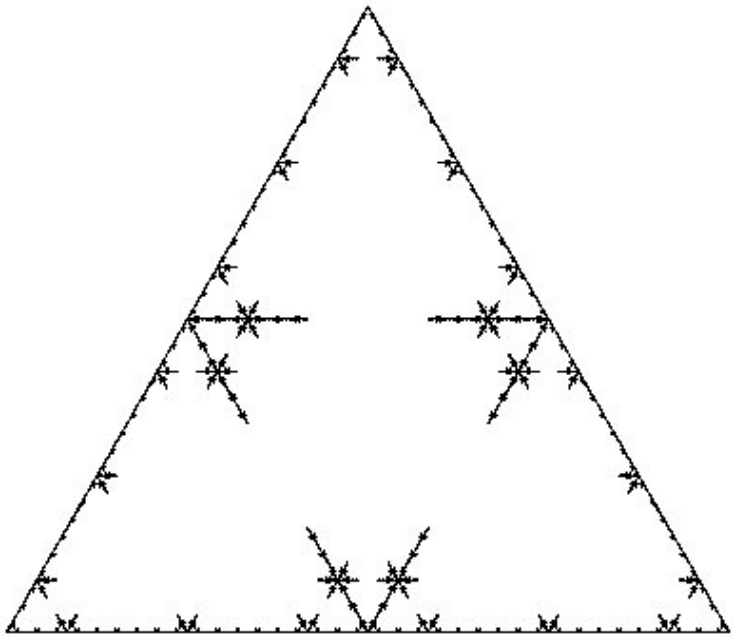
b. Fractal de Hielo, basado en un triángulo,
$B \rightarrow IA_nDDA_nDDA_n$, con un ángulo de 60° para formar el triángulo equilátero,
$A_0 \rightarrow$ Avanzar,
$A_{n+1} \rightarrow A_nA_nA_nIIA_nDDDA_nIIA_nDDDA_nIIA_nA_nA_n$,
Todos los giros se hacen con un ángulo de 60° (= π/3 radianes).
Resolución: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

Calcule A_4.



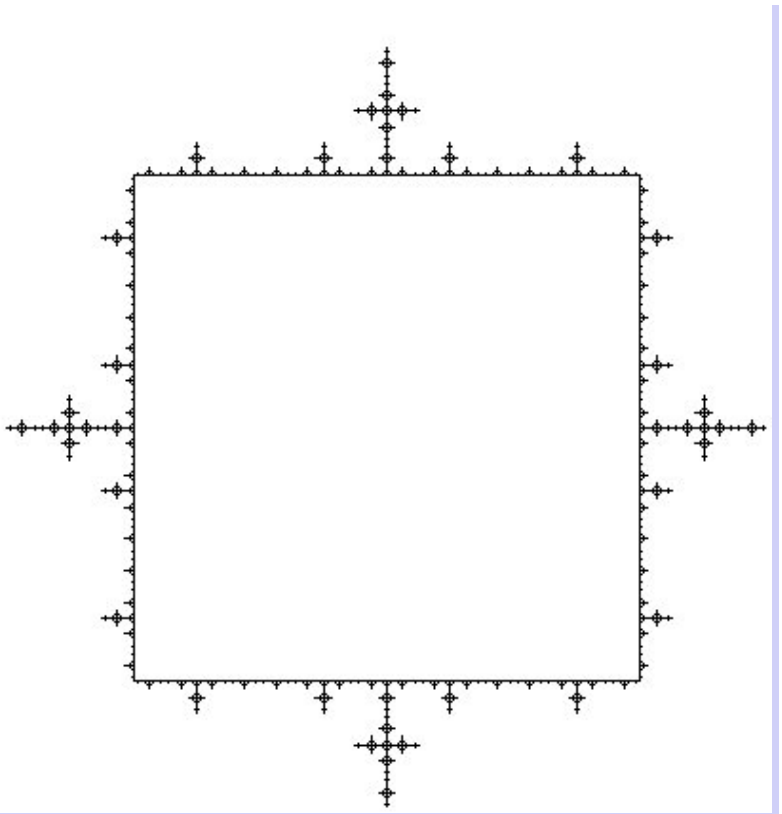
Fractal de Hielo - N=4

c. Fractal de Hielo, basado en un triángulo,
$B \rightarrow A_n IIA_n IIA_n$, 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.



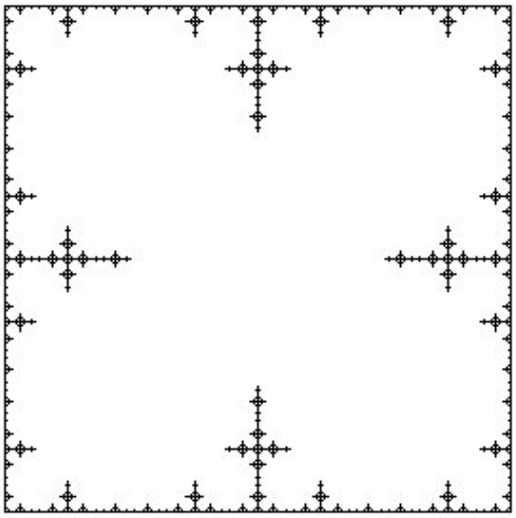
Fractal de Hielo - N=4

d. Fractal de Hielo, basado en un cuadrado,
$B \rightarrow A_n I A_n I A_n I A_n$, con un ángulo de 90° (=π/2 radianes),
describe un cuadrado regular,
$A_0 \rightarrow$ Avanzar,
$A_{n+1} \rightarrow A_n A_n I A_n D D A_n I A_n A_n$,
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.
Calcule A_4.



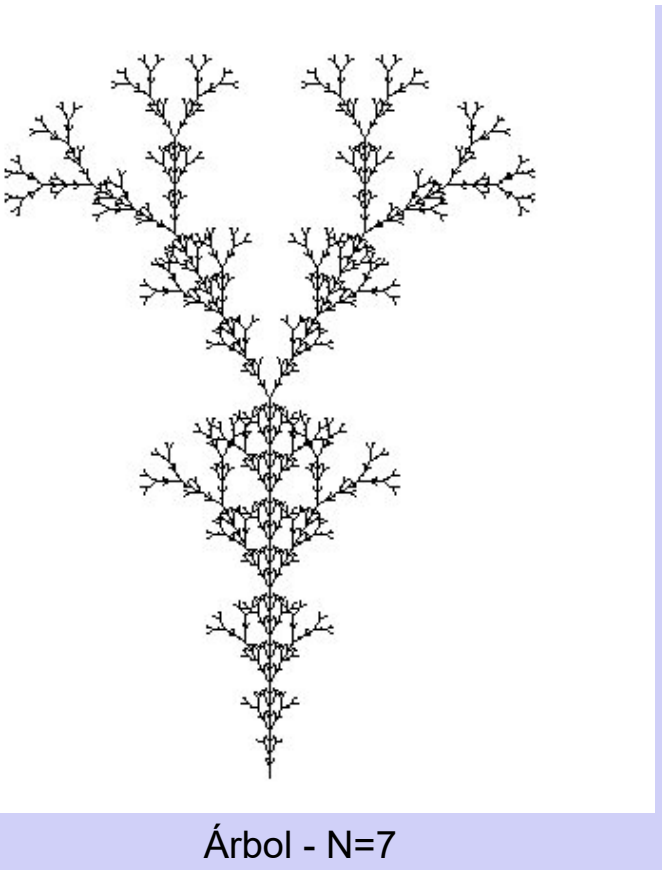
Fractal de Hielo - N=4

e. Fractal de Hielo, basado en un cuadrado,
   $B \rightarrow A_nDA_nDA_nDA_n$, 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,
$A_0 \rightarrow$ Avanzar,
$A_{n+1} \rightarrow A_nA_n[IA_n][DA_n]$,
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.
Calcule A_7.



Árbol - N=7

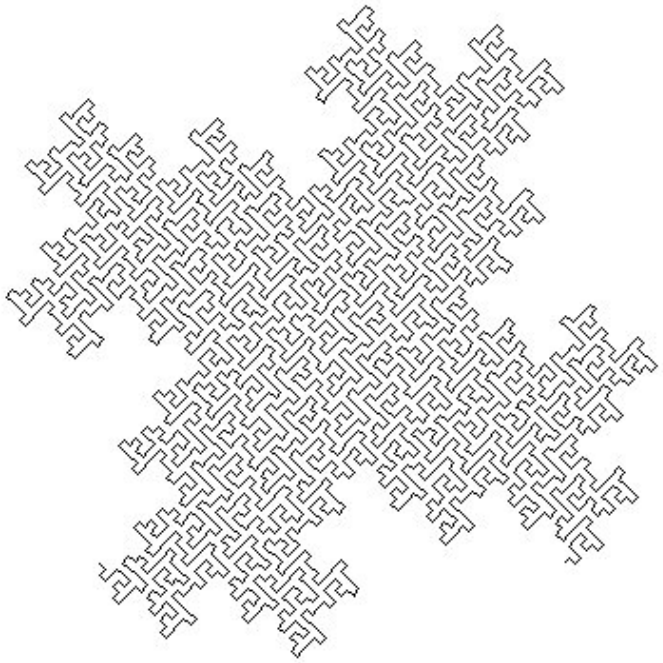
6. Ahora agregaremos otro símbolo a nuestra gramática. Se trata de la letra **E** que indica avanzar en el tramo que se encuentra pero en forma de espejo. Tenemos que invertir las instrucciones de la regla al igual que cambiar todos los casos de **I**zquierda a **D**erecha y de **D**erecha a **I**zquierda. Por ejemplo, si tenemos la siguiente regla:
   $A_0 \rightarrow$ Avanzar,
   $E_0 \rightarrow$ Avanzar,
   $A_{n+1} \rightarrow IE_nA_nDA_nDE_nIA_n$, con un ángulo de giro de 90° (= π/2 radianes), y
   $E_{n+1} \rightarrow A_nDE_nIA_nIA_nE_nD$

   Como pueden 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 $A_0DE_0IA_0IA_0E_0D$

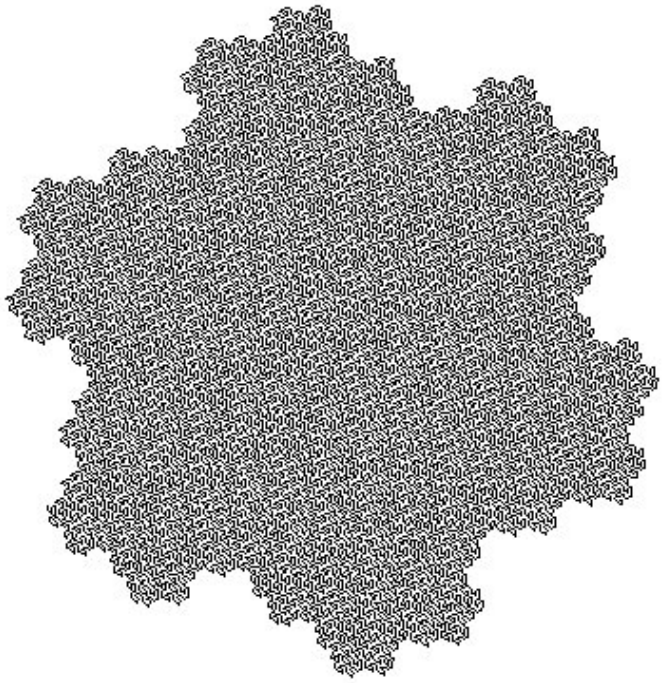
7. Cree un programa para que trace cada uno de los siguientes fractales:
   a. Curva basada en Peano,
      $A_0 \rightarrow$ Avanzar, y
      $A_{n+1} \rightarrow IE_nA_nDA_nDE_nIA_n$, con un ángulo de giro de 90° (= π/2 radianes)
      ángulo inicial: -132,825° (= -2,3182 radianes) o si lo prefieren: 227,175° (= 3,9650 radianes).
      Resolución: 400x400.
      Longitud original (del Avance): 231 píxeles
      División del tramo: d=√(1/5) = 0,44721.
      Calcule A_5.



Curva basada en Peano - N=5

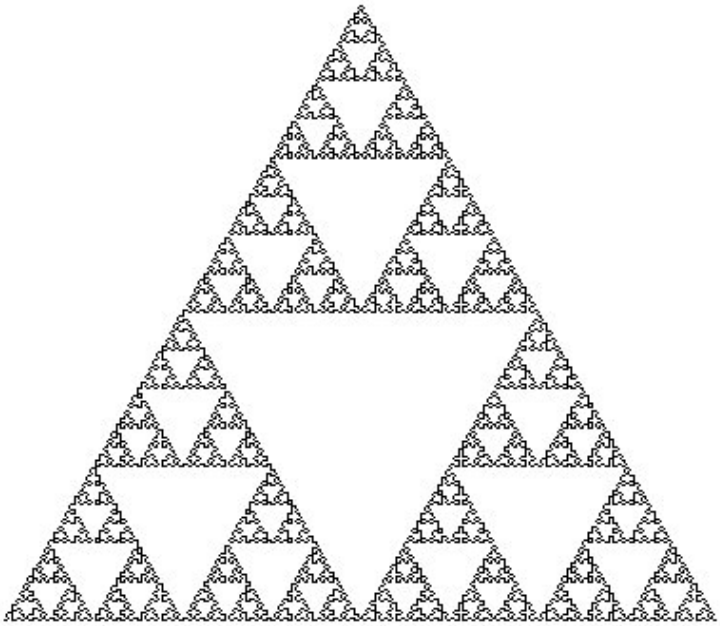
   b. Curva de Peano-Gosper,
      $A_0 \rightarrow$ Avanzar,
      $A_{n+1} \rightarrow A_nIE_nIIE_nDA_nDDA_nA_nDE_nI$,

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.
Calcule A_5.



Curva de Peano-Gosper - N=5

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



Curva de Peano-Sierpinski - N=8

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*.

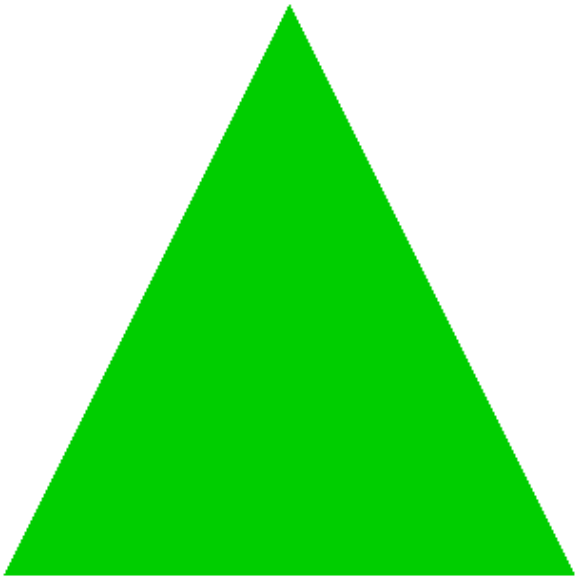


Figura 1 - Sierpinski n=0

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*.

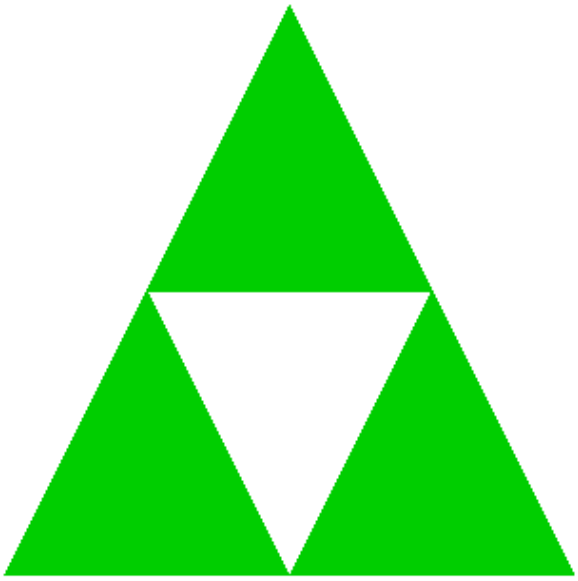


Figura 2 - Sierpinski n=1

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*.

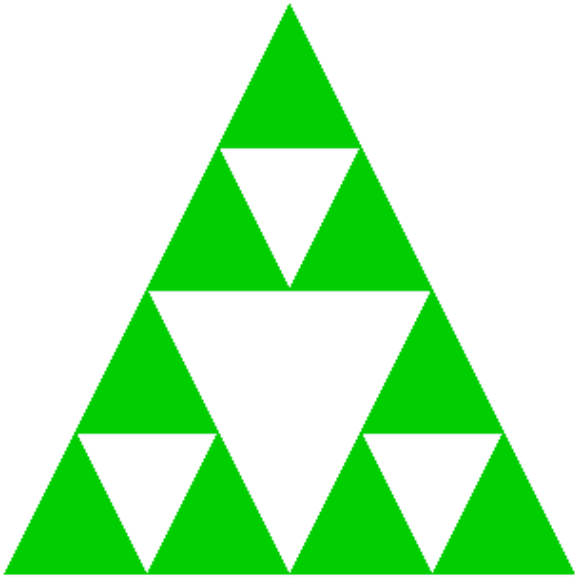


Figura 3 - Sierpinski n=2

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:

```
Iniciar_Sierpinski()

 1. Declarar:  A, B, y C  como puntos
 2. Inicializar:  A, B, y C
 3. Dibujar_Sierpinski( A, B, C, Número_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_Triángulo_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_Triángulo_Relleno()* hace referencia a una función de la biblioteca 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:

$$d_{cap} = - \lim_{n \to \infty} \frac{\ln N_n}{\ln L_n} = - \lim_{n \to \infty} \frac{\ln 3^n}{\ln 2^{-n}} = - \lim_{n \to \infty} \frac{n * \ln 3}{-n * \ln 2} =$$

$$= \lim_{n \to \infty} \frac{\ln 3}{\ln 2} = \frac{\ln 3}{\ln 2} = 1{,}584962501...$$

(Nota: ∞ 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.

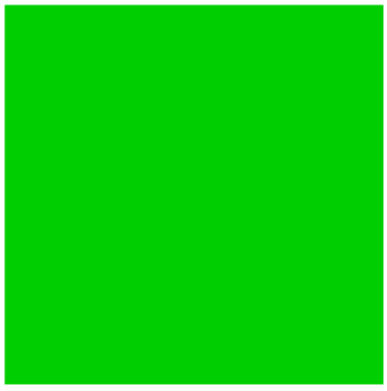
Ejercicios

Enlace al paquete de este capítulo.

1. Escriba un programa que dibuje el triángulo de Sierpinski. Se puede usar el algoritmo presentado en este capítulo. Use las siguientes restricciones y condiciones iniciales:
   Resolución: 500 x 500.
   Calcule, con N=5 iteraciones.
2. El triángulo de Sierpinski no tiene por qué ser equilátero. Construya el triángulo de Sierpinski para cada triángulo descrito por sus vértices, use las siguientes restricciones:
   Resolución: 500 x 500
   Calcule, 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. Escriba 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 seleccionando 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. A este fractal se le denomina *Alfombra de Sierpinski*. La siguiente figura puede servir de ejemplo:



Alfombra de Sierpinski N=[0,4]

Escriba un programa para representar la alfombra de Sierpinski con las siguientes restricciones y valores iniciales:

Resolución: 500 x 500.
Calcule, con N=4 iteraciones.

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 500 x 500. 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:

```
Iniciar_Mandelbrot()
```

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

```
      num_max_iteraciones, radio, C
 2. anchura_u Fracta |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_up ← altura_u / altura_p
 8. Bucle: x_p ← x_pi hasta x_pf con incremento de 1

         9. Bucle: y_p ← y_pi hasta y_pf con incremento de 1

                10. C.x ← x_ui + x_p * dx_up
                11. C.y ← y_ui - y_p * dy_up
                12. Color ← Mandelbrot( C,
                    num_max_iteraciones, radio )
                13. PonPíxel( x_p, y_p, Color )


14. Terminar
```

- *C* 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 *PonPíxel()* hace referencia a una función de la biblioteca 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 ← Ecuación( Z, C )
 4. Bucle: k ← 1 hasta max_iter  con incremento de 1 y
           mientras que  bEsMandelbrot = Verdadero

         5. Si Distancia( Z_Sig ) < r  entonces

                6. Z ← Z_Sig
                7. Z_Sig ← Ecuación( Z, C )
```

```
          8. Si no, entonces

                  9. bEsMandelbrot ← Falso

10. Si bEsMandelbrot = Verdadero, entonces

        11. Terminar( Negro )

12. Si no, entonces

        13. Terminar( Blanco )
```

C, *Z*, y *Z_Sig* pueden almacenar una coordenada en unidades cartesianas, cada una.

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 *Ecuación()*, el algoritmo es:

```
Complejo Ecuación( 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 periodicidad 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 estéticamente 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 á 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. Escriba 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. Realice un programa que genere un fractal basado en cada una de las siguientes fórmulas, reemplazando el algoritmo *Ecuación()* en el apartado **Algoritmo**.

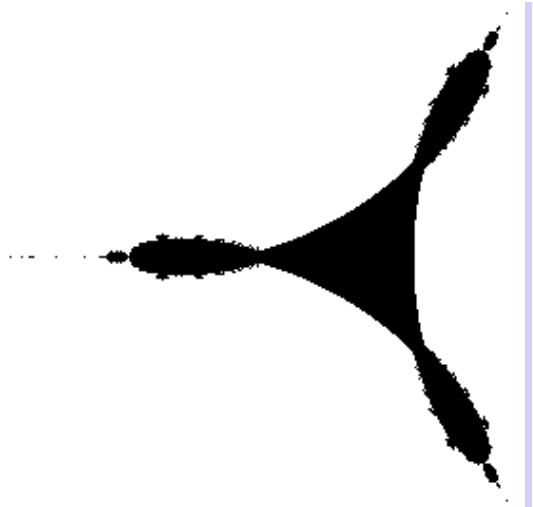
   Nota: Se aconseja usar alguna biblioteca para manipular números complejos, especialmente si no se sabe hacer usando matemáticas. En un programa de C++, se puede usar la plantilla `complex<T>` declarada en `<complex>`. Se sugiere usar la clase `complex<double>` para crear cada variable compleja y usar 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:

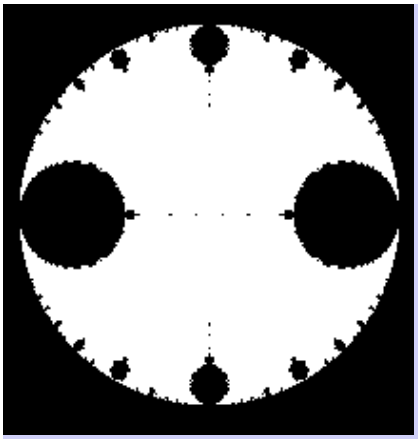
      z = x + iy o z = (x,y),

su conjugado es:
z' = x - iy o z = (x,-y),



Ejercicio 2a

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

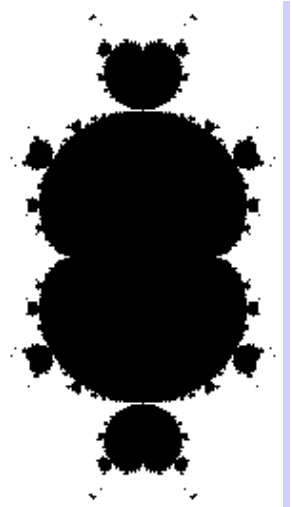


Ejercicio 2b

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

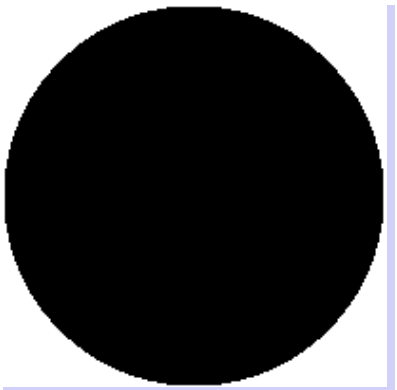
Ejercicio 2c

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



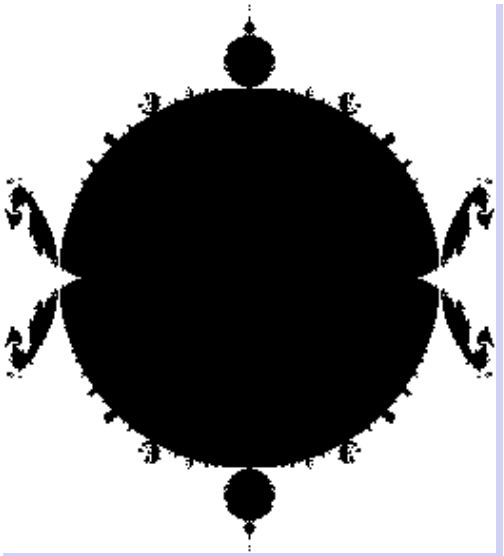
Ejercicio 2d

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



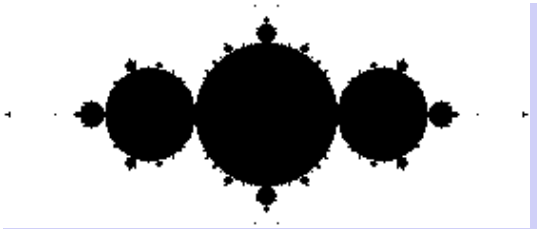
Ejercicio 2e

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



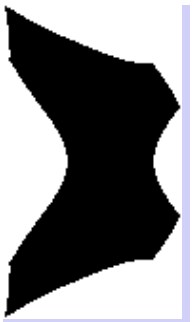
Ejercicio 2f

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



Ejercicio 2g

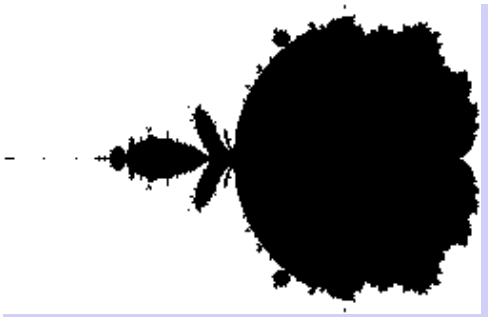
h. $Z_{n+1} = C*exp(conj(Z_n)) * C*exp(Z_n) = C^2*exp(2x)$



Ejercicio 2h

Nota: x indica la parte real de Z_n

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



Ejercicio 2i

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

Ejercicio

o
2j

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

Ejercicio 2k

l. $Z_{n+1} = \ln(Z_n) + Z^2 + C$

Nota: `ln` hace alusión al logaritmo neperiano o natural

Ejercicio 2l

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 bibliotecas estándares, las implementaciones de algunas funciones no den buenos resultados. Por lo tanto, use las siguientes definiciones:

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. Cree 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 unas 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 se 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)]
   y = [0-0, +2-0]

   Obtenemos,
   x = [0, +4]
   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 original por 3,0, que es lo mismo que dividir las longitudes entre 3,0. Esto sería:
   x = [0/3,0, 4/3,0]
   y = [0/3,0, 2/3,0]

   Resultando en,
   x = [0,0, 1,3333]
   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]

y = [-0,5, +0,1666]

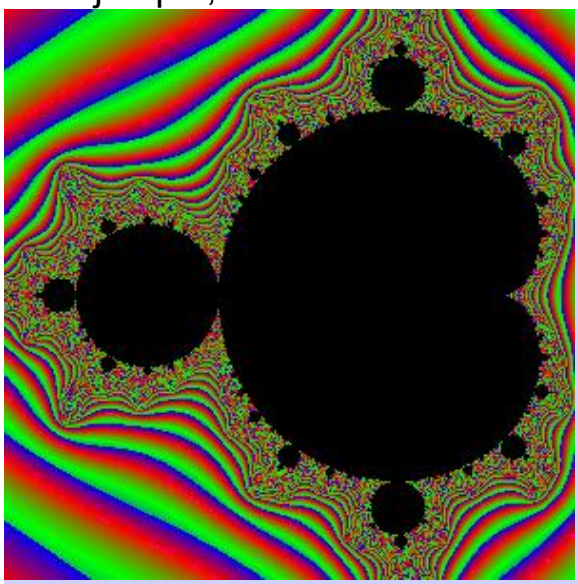
Recalculando el fractal con estas nuevas dimensiones, obtendremos una imagen de un área reducida 3 veces de la original y centrada en el punto (0,5, -0,5). Como el área es más pequeña que la original, los valores de $\mathbf{dx_{up}}$ y $\mathbf{dy_{up}}$ se ven reducidos. Esto implica que nuestra imagen permitirá mostrar más información al someter estas nuevas dimensiones a la misma resolución gráfica.

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.
Por ejemplo,



Ejercicio 6

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, 500 x 500, entonces dicha imagen tiene un tamaño de 500 × 500 × 15 = 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 500 x 500 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 un intervalo de 0 á 255 (ambos incluidos), cada una. 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 máquina 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: 500 x 500 x 16, 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, 500 × 500 × 8 = 2.000.000 bits = 250.000 bytes, que es aproximadamente, 244 KB. Por supuesto, tenemos que agregar la paleta de 256 colores × 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.

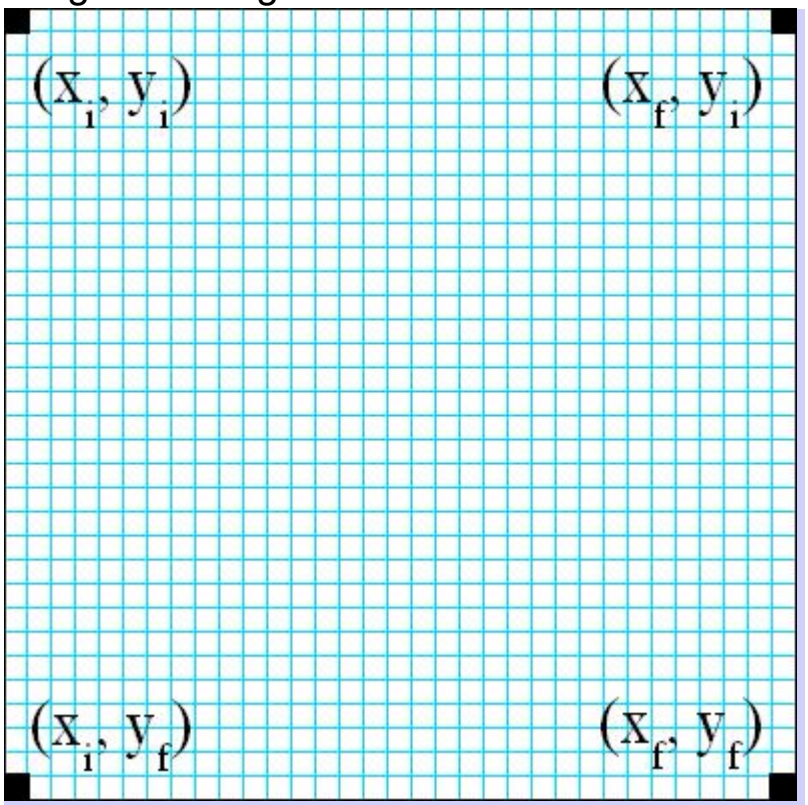


Figura 1 - Las 4 esquinas

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:

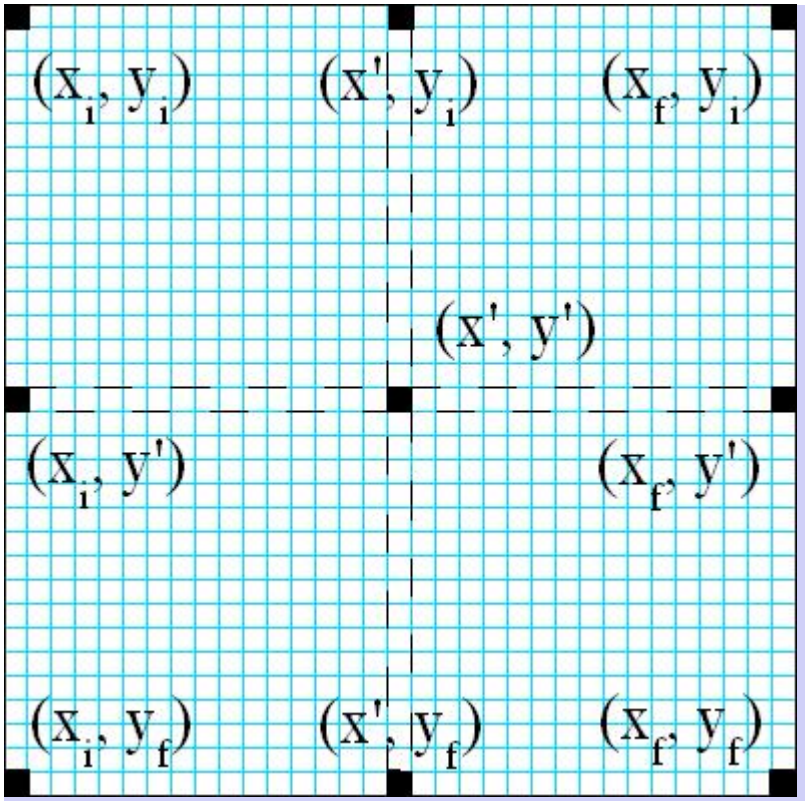


Figura 2 - 9 puntos totales

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:

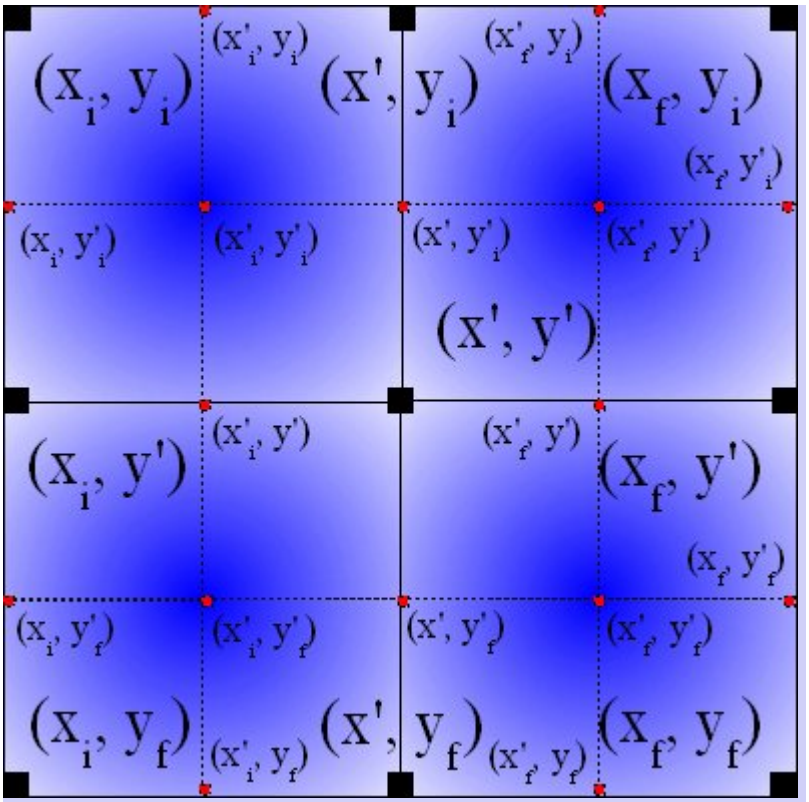


Figura 3 - Siguiente Iteración

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

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:

```
Generar_Nube()

 1. Iniciar: valor_semilla, anchura_p, altura_p
 2. Inicializar: parámetros[anchura_p, altura_p] ← -1 o
    cualquier valor negativo
 3. Iniciar_Generador_Números_Aleatorios( valor_semilla )

    // Obtener parámetros de colores
 4. parámetros[0,0] ← Generar_Número(256)
 5. parámetros[anchura_p-1,0] ← Generar_Número(256)
 6. parámetros[anchura_p-1,altura_p-1] ← Generar_Número(256)
 7. parámetros[0,altura_p-1] ← Generar_Número(256)

    // Dibujar Esquinas
 8. PonPíxel( 0,0, Mapa( parámetros[0,0] ) )
```

```
 9. PonPíxel( anchura_p-1,0, Mapa( parámetros[anchura_p-1,0] )
    )
10. PonPíxel( anchura_p-1,altura_p-1, Mapa(
    parámetros[anchura_p-1,altura_p-1] ) )
11. PonPíxel( 0,altura_p-1, Mapa( parámetros[0,altura_p-1] ) )

    // Comenzar la Recursividad
12. Subdividir( parámetros, 0,0, anchura_p-1,altura_p-1 )
13. 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, que indican un estado inválido.
- 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 *PonPíxel()* 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  Y  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. PonPíxel( 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. PonPíxel( 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. PonPíxel( 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. PonPíxel( 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. PonPíxel( 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
    derecho
22. Subdividir( matriz, x_i,y, x,y_f )   // Cuadrante inferior
    izquierdo
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).

Para *Generar_Parámetro()*, el algoritmo es:

```
Generar_Parámetro( parámetros, x_a,y_a, x,y, x_b,y_b )

 1. t ← |x_a-x_b| + |y_a-y_b|
 2. t ← Generar_Número(2*t) - t +
         ( parámetros[x_a,y_a] + parámetros[x_b, y_b] + 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:

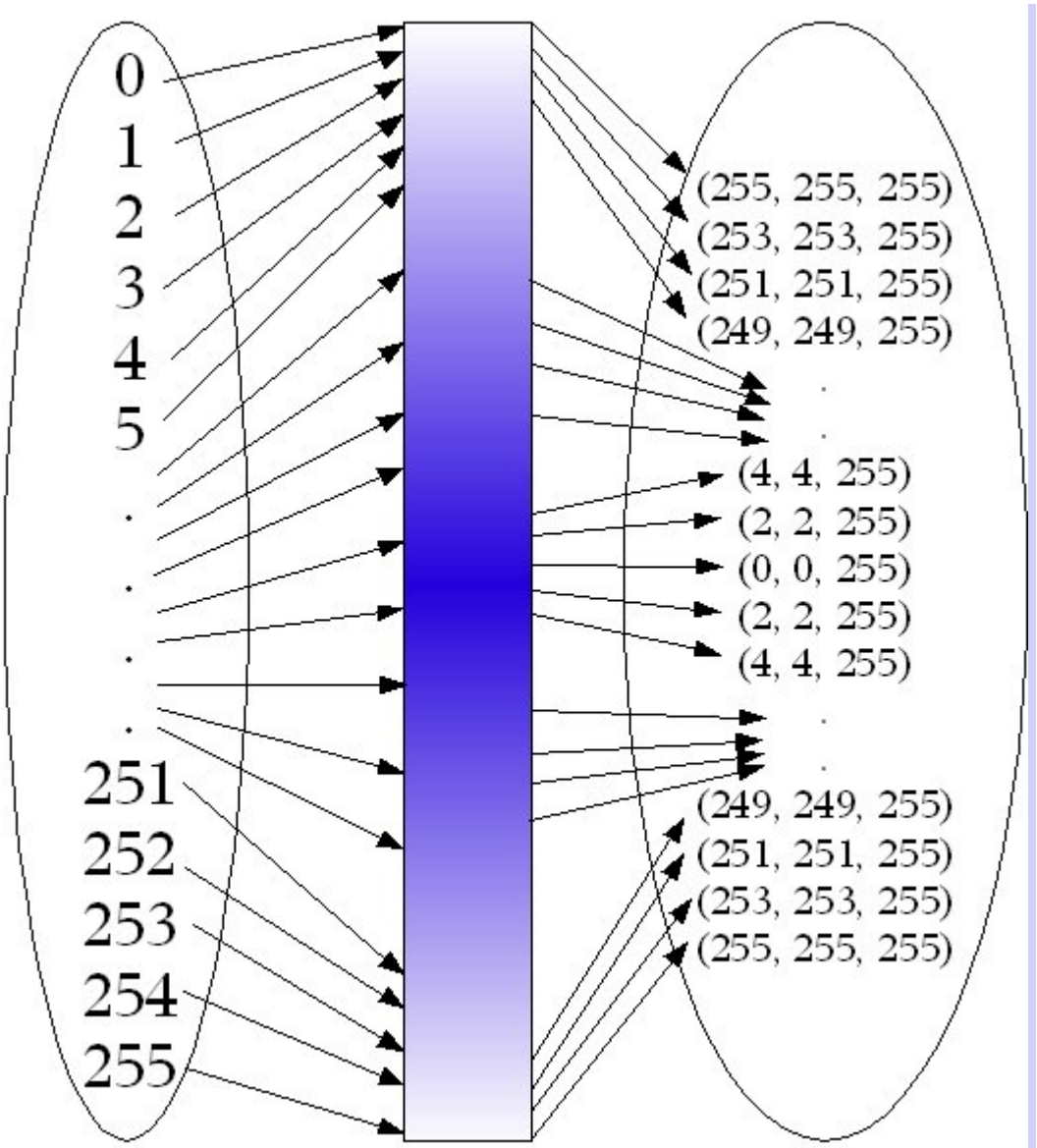


Figura 5 - Mapa: Parámetro-Color

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.

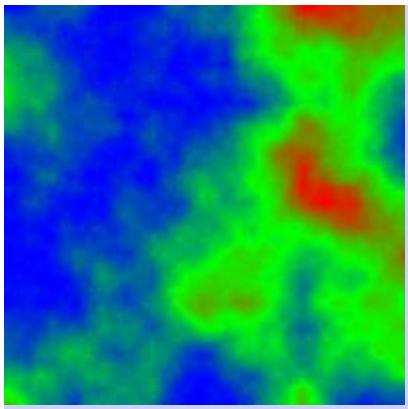
Ejercicios

Enlace al paquete *de este capítulo.*

1. Escriba un programa que implemente el algoritmo descrito en este capítulo para generar nubes. Use una resolución de 400 x 400.
2. Escriba un programa usando el mapa de colores descrito en cada apartado más abajo, con una resolución de 400 x 400.
   a. Plasma:

```
Color Mapa( x )
```

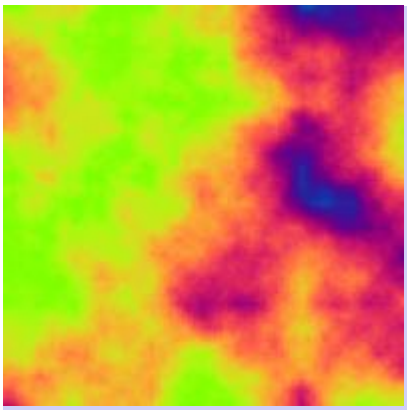
```
 1. Si 0 ≤ x ≤ 85

         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 )
```



Ejercicio 2a

b. Fuego-Hielo:

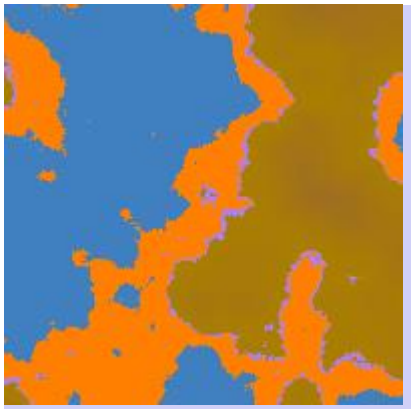
```
Color Mapa( x )
 1. color.rojo  ← 255*(1+sen(2*π/256*x))/2
 2. color.verde ← 255*(1+cos(2*π/256*x))/2
 3. color.azul  ← x
 4. Terminar( color )
```



Ejercicio 2b

c. Topográfico:

```
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 ← 318-x  // De 255 á 63
    11. color.rojo  ← 255-x/3
    12. color.verde ← 255-x/2
    13. color.azul  ← x
  14. Terminar( color )
```



Ejercicio 2c

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 Interpolación( yi, yf, x, xi, xf )
 1. y ← yi*(x-xf)/(xi-xf) + yf*(x-xi)/(xf-xi)
 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,

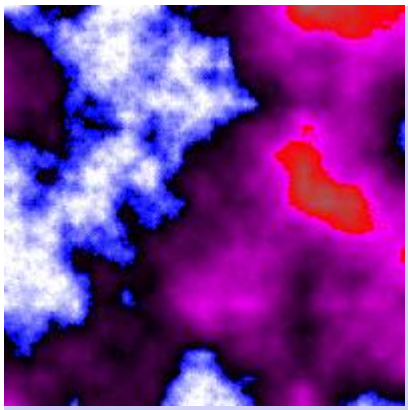
$$y = 5 * \frac{(3 - 2)}{(4 - 2)} + 10 * \frac{(3 - 4)}{(2 - 4)} = 5/2 + 10/2 = 7,5$$

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



 9. Si  0,50 < i

                // De Rojo a Amarillo

        10. color.rojo  ← 255*(1,00-i)/0,50

        11. color.verde ← 255*(i-1,00)/0,50

        12. color.azul  ← 255*(i-0,50)/0,40



13. Terminar( color )
```



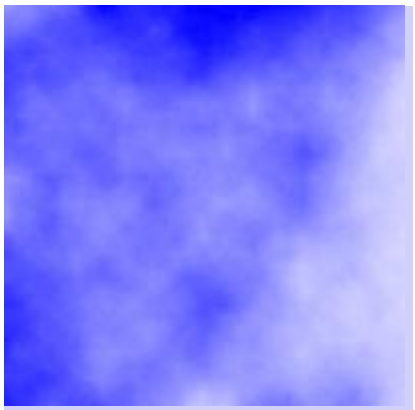
Ejercicio 3

Cree 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 desee usar. Luego, al invocar el proceso, pasamos un puntero a la función con el mapa de colores que nos interesa.

   Cree más mapas y siga 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. Aplique 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 ) )
```

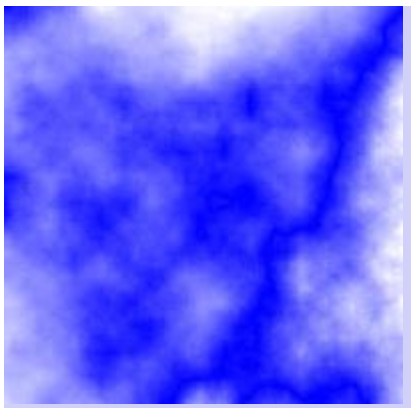


Ejercicio 5a

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

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

donde, `e = 2,7182818285` y la función exponencial: $\exp(x) = e^x$



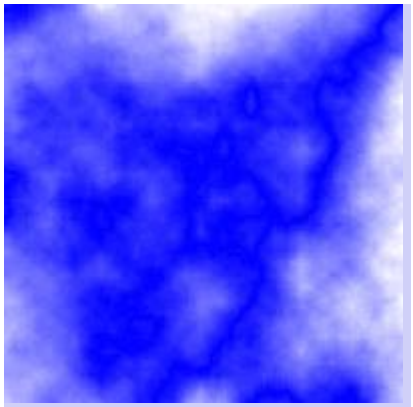
Ejercicio 5b

Observación: Aplicando una función exponencial es una técnica para suavizar una imagen. Compare la imagen de nubes producida con el método descrito en este capítulo y la imagen generada aplicando la función exponencial. En

general, se notará una imagen más suave y algo borrosa generada con la función exponencial.

c. Aplique la función del seno hiperbólico al parámetro. Si el parámetro se encuentra en el intervalo: [0,00, 1,00], entonces use la siguiente fórmula:

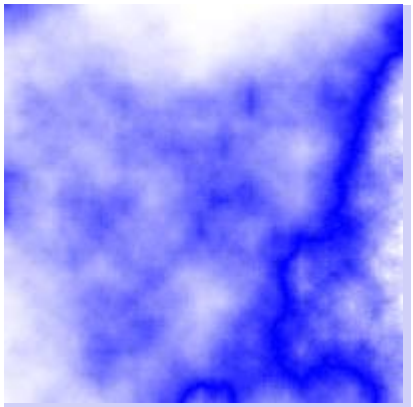
```
parámetro ← senh(matriz[x,y]) / senh(1)
PonPíxel( x,y, Mapa( parámetro ) )
```



Ejercicio 5c

d. Aplique la función del coseno hiperbólico al parámetro. Si el parámetro se encuentra en el intervalo: [0,00, 1,00], entonces use la siguiente fórmula:

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



Ejercicio 5d

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

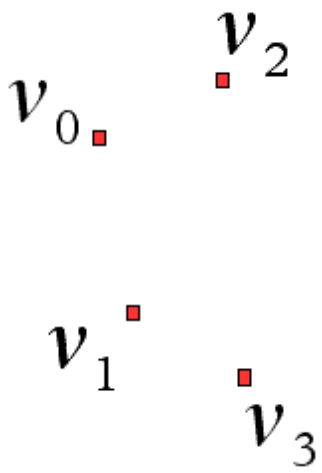


Figura 1 - Vértices

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.

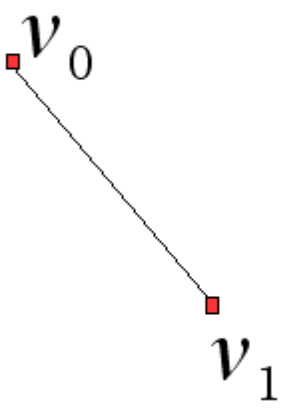


Figura 2 - Línea

Triángulo

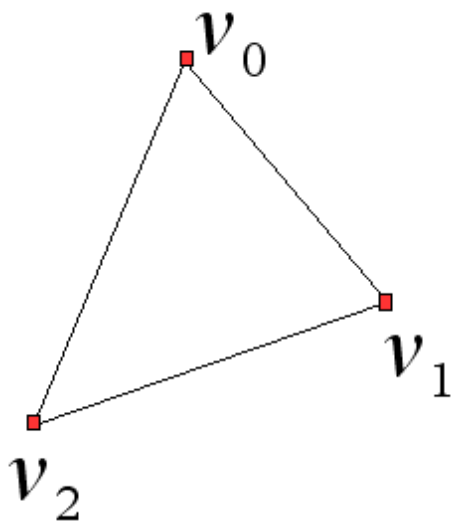


Figura 3 - Triángulo

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.

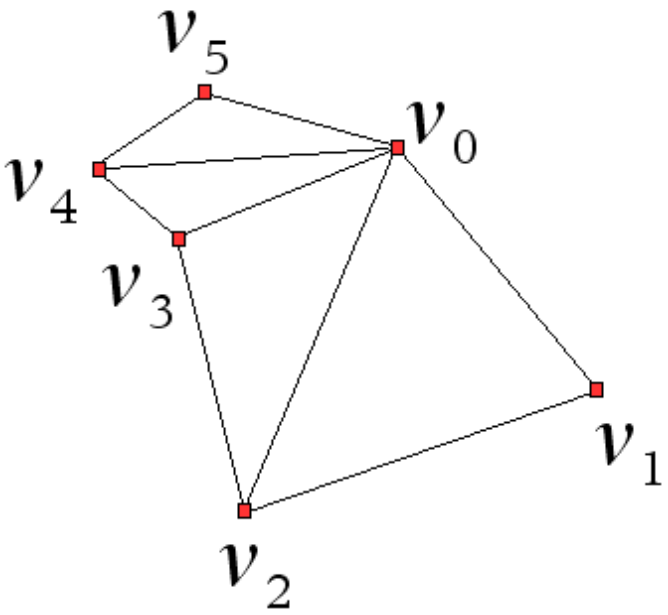
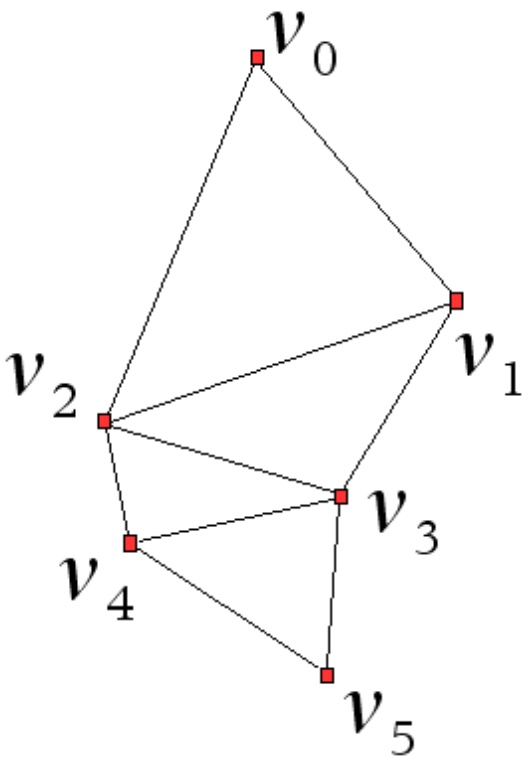


Figura 4 - Abanico de triángulos

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.

Figura 5 - Tira de triángulos

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.

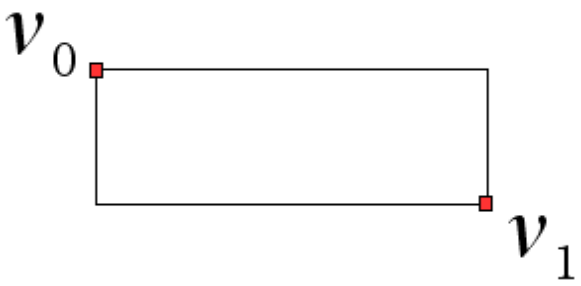


Figura 6 - Rectángulo

Cuadrilátero

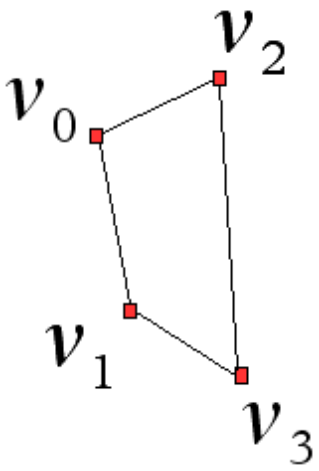


Figura 7 - Cuadrilátero

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.

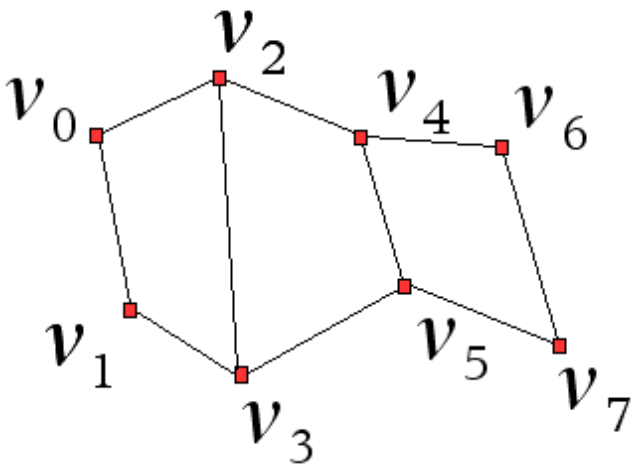


Figura 8 - Tira de cuadriláeros

Polígonos

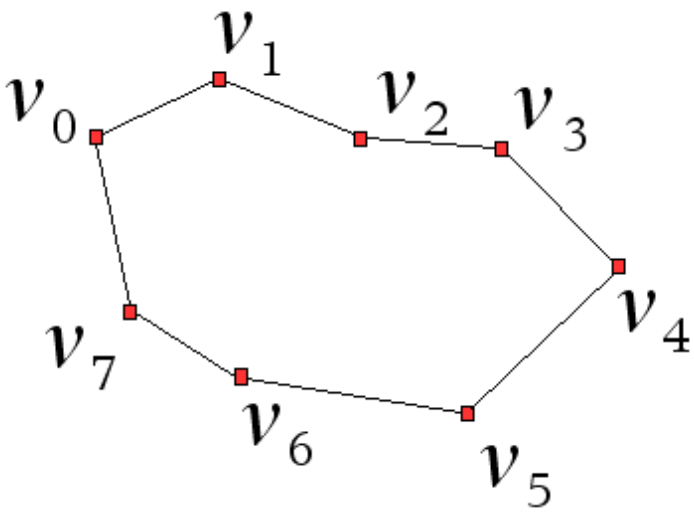


Figura 9 - Polígono

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. Algunas API's gráficas describen las elipses y círculos de formas diferentes. Existen principalmente dos maneras de describir tales figuras:

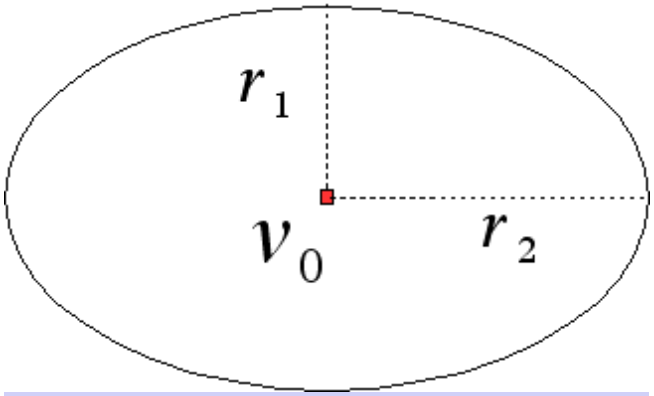


Figura 10.1 - Elipse/Círculo #1

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.

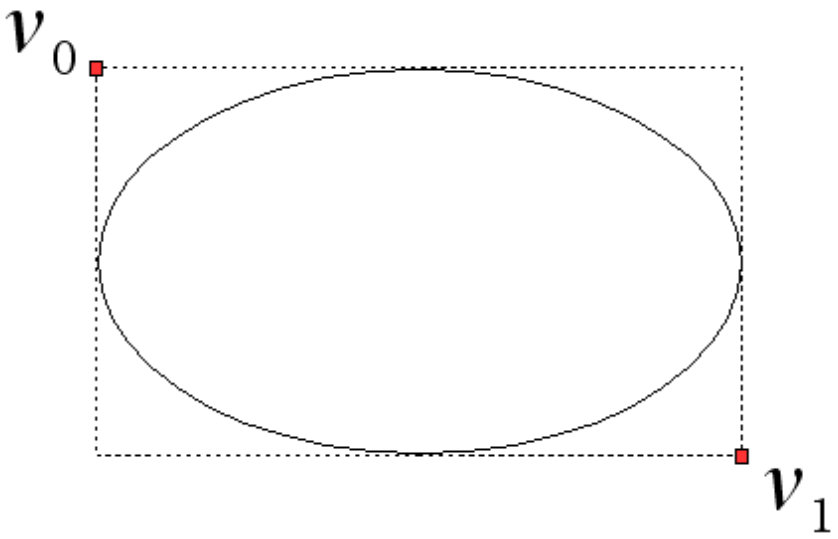


Figura 10.2 - Elipse/Círculo #2

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:

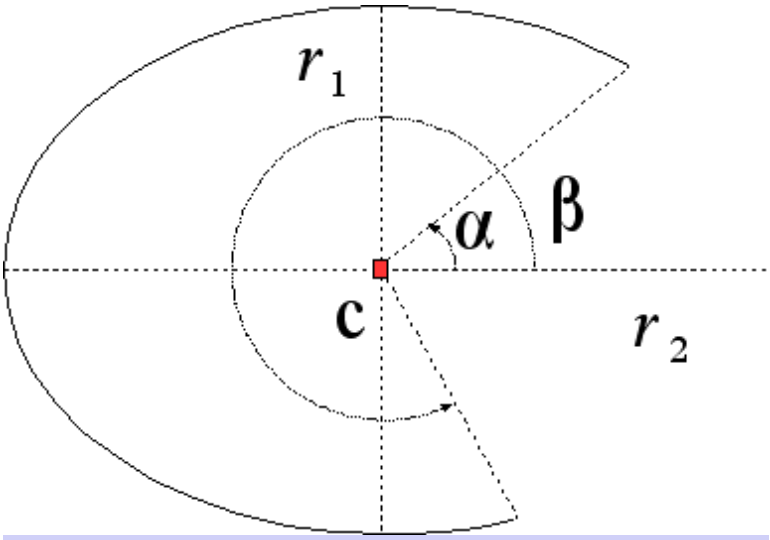


Figura 11.1 - Arco #1

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*.

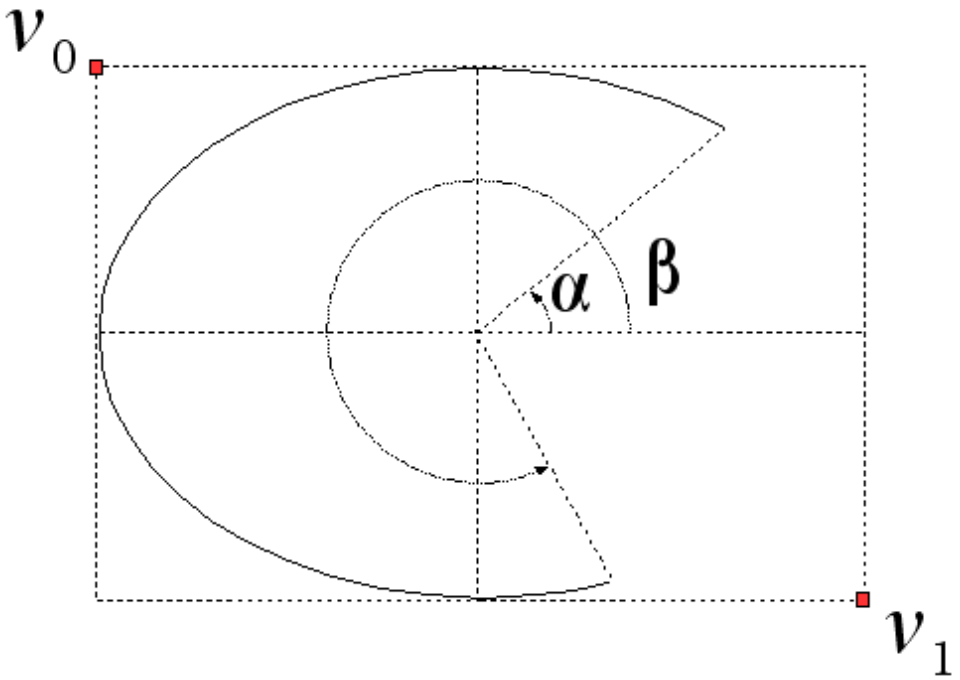


Figura 11.2 - Arco #2

Sector/Cuña

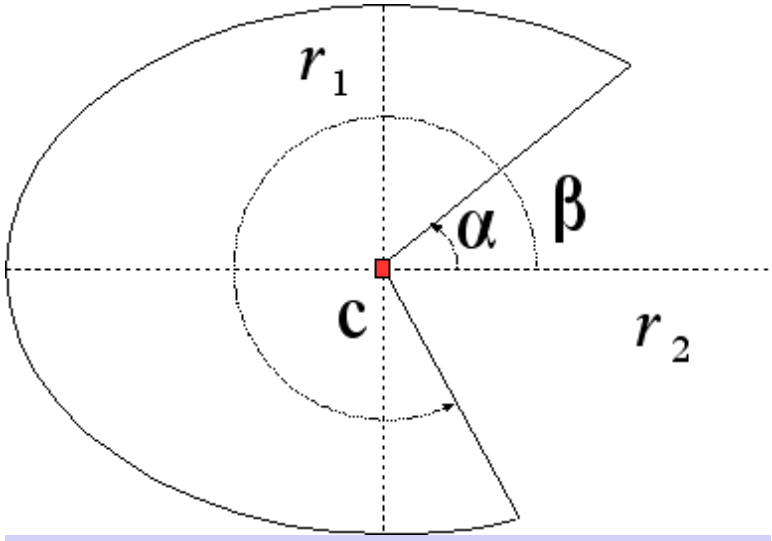


Figura 12.1 - Sector/Cuña #1

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.

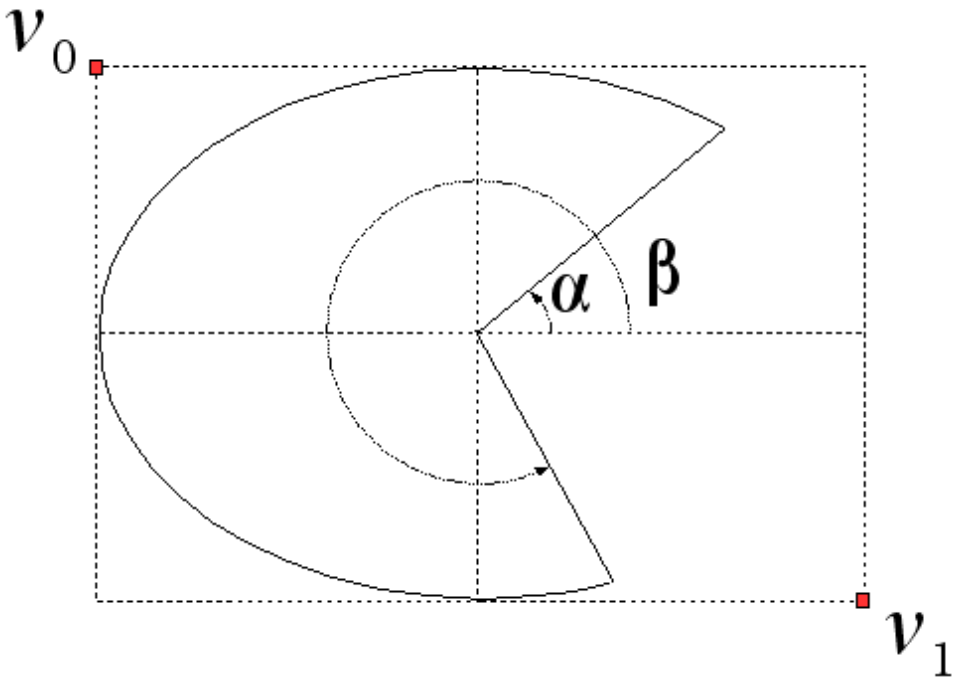


Figura 12.2 - Sector/Cuña #2

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:

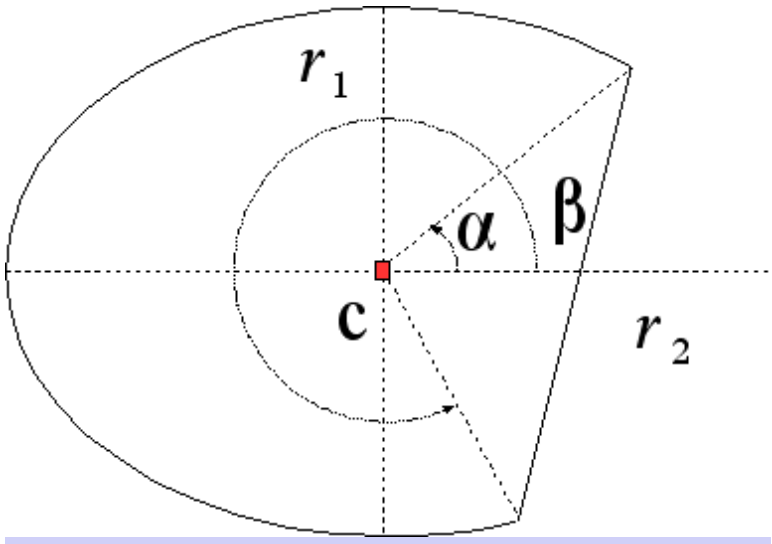


Figura 13.1 - Sector elíptico #1

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*.

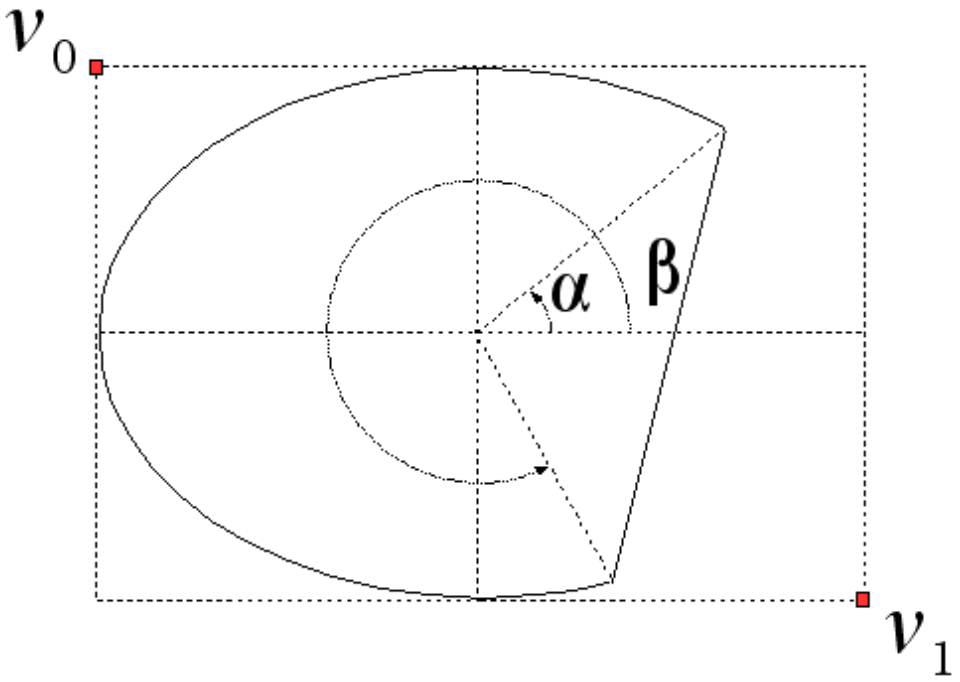


Figura 13.2 - Sector elíptico #2

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.

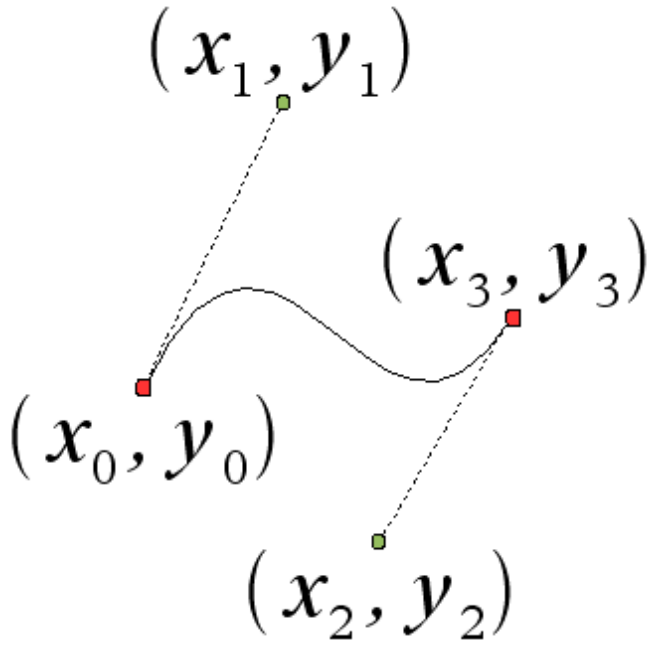


Figura 14 - Curva de Bézier

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

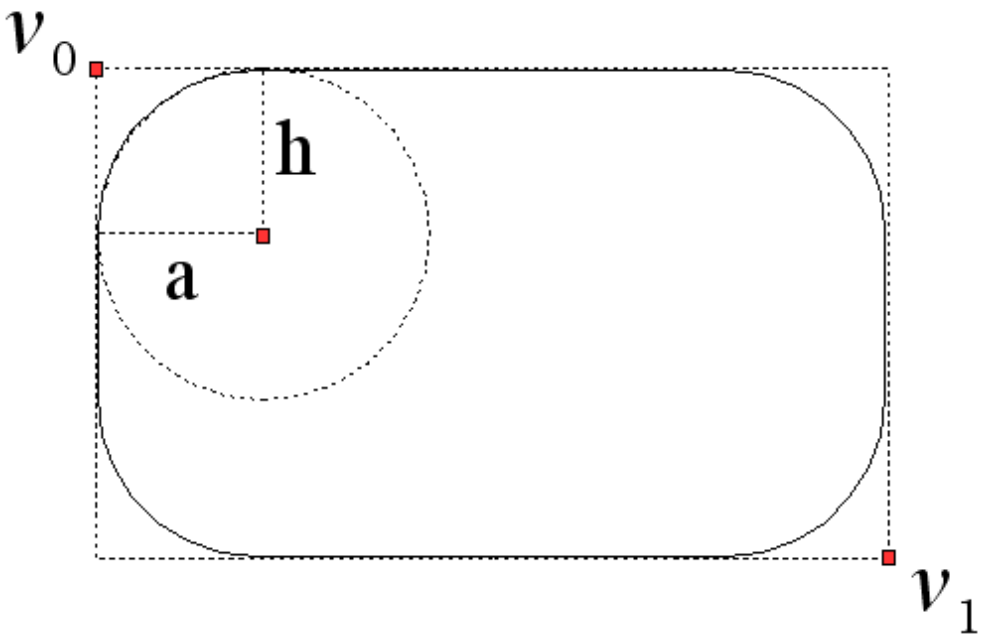


Figura 15 - Rectángulo redondo

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 2π = 360°. En otras palabras, cada ángulo mencionado es inversamente proporcional a la cantidad de lados del polígono regular. Podemos usar la siguiente fórmula:

$\alpha = 2\pi / N$, donde α es el ángulo, y N 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:

- $i = 0,1,2,\ldots,N-1$,
- 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

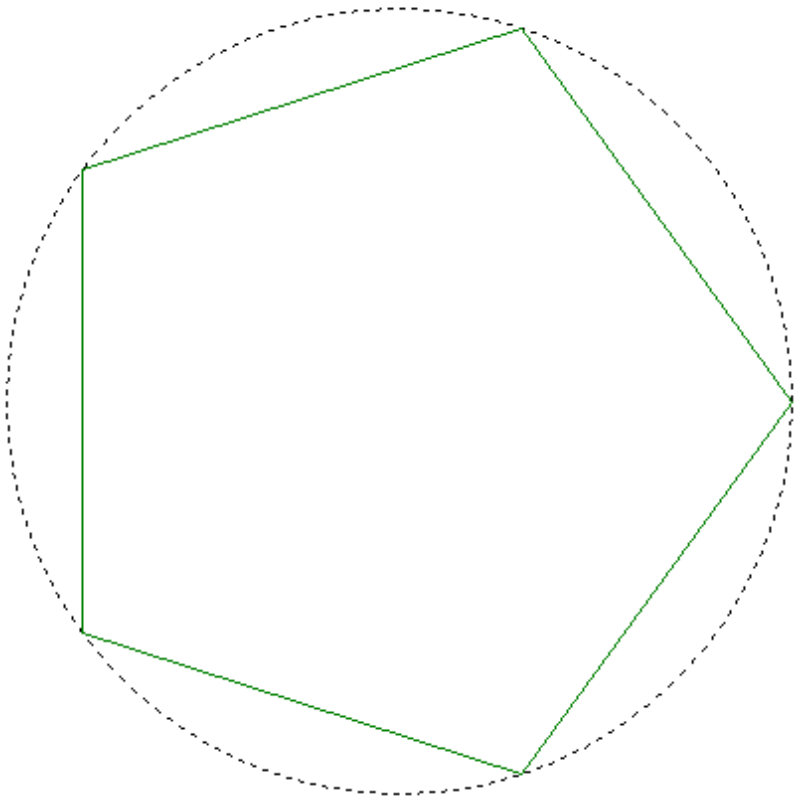


Figura 16 - Polígono - N=5

Con un pentágono, tenemos que N=5, por lo que α = 2π / 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.

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 Figura 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 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_Polígono()*:

```
Dibujar_Polígono( vértices, N )

  1. Bucle: i ← 2 hasta N

        2. Dibujar_Polígono( vértices[i-1].x, vértices[i-
           1].y, vértices[i].x, vértices[i].y )


  3. Dibujar_Polígono( 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,

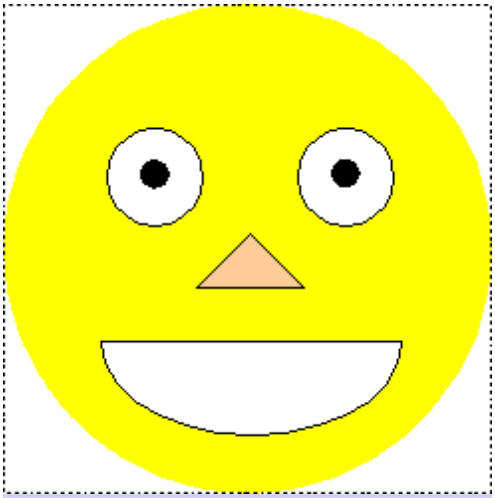


Figura 17 - Cara

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.

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.

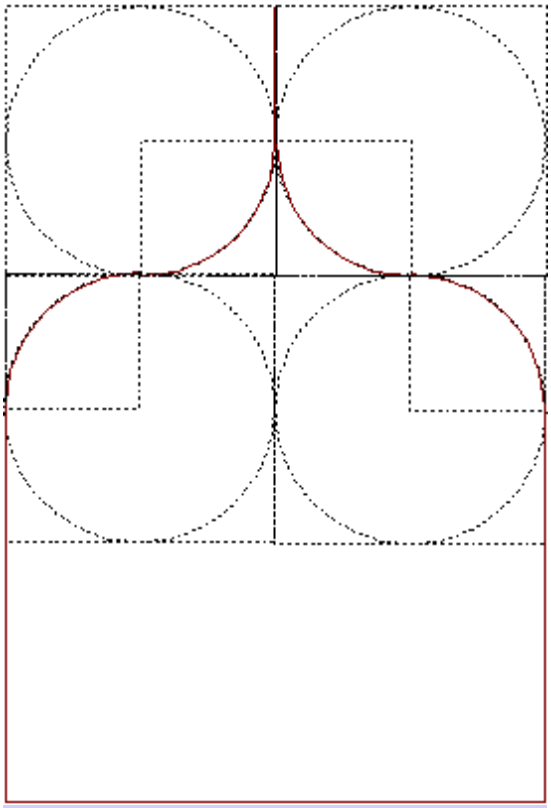


Figura 18 - Arco Persa

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

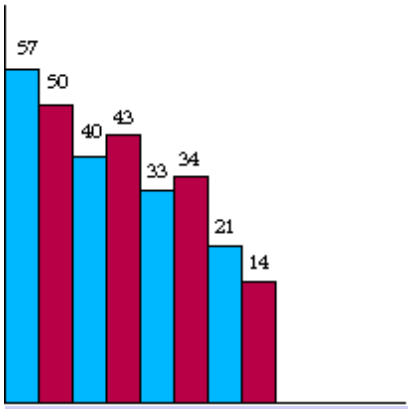


Figura 19 - Barras

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.

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.

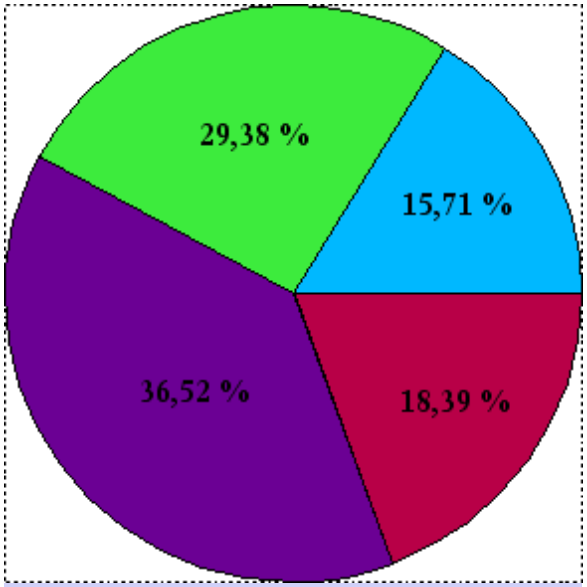


Figura 20 - Gráfico Circular

4. Diagramas

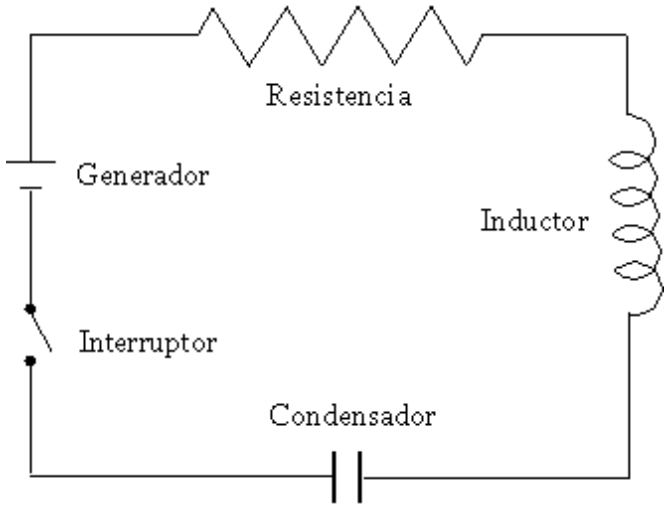


Figura 21 - Circuito Eléctrico

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

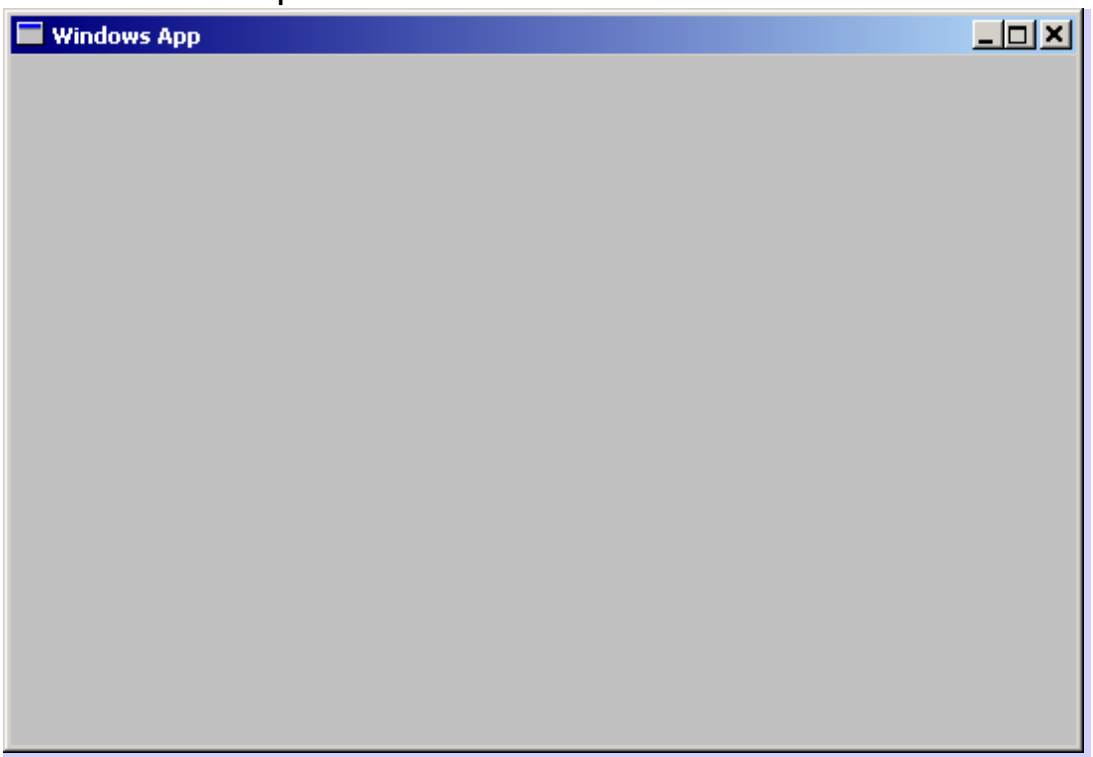


Figura 22 - Ventana de MS-Windows

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.

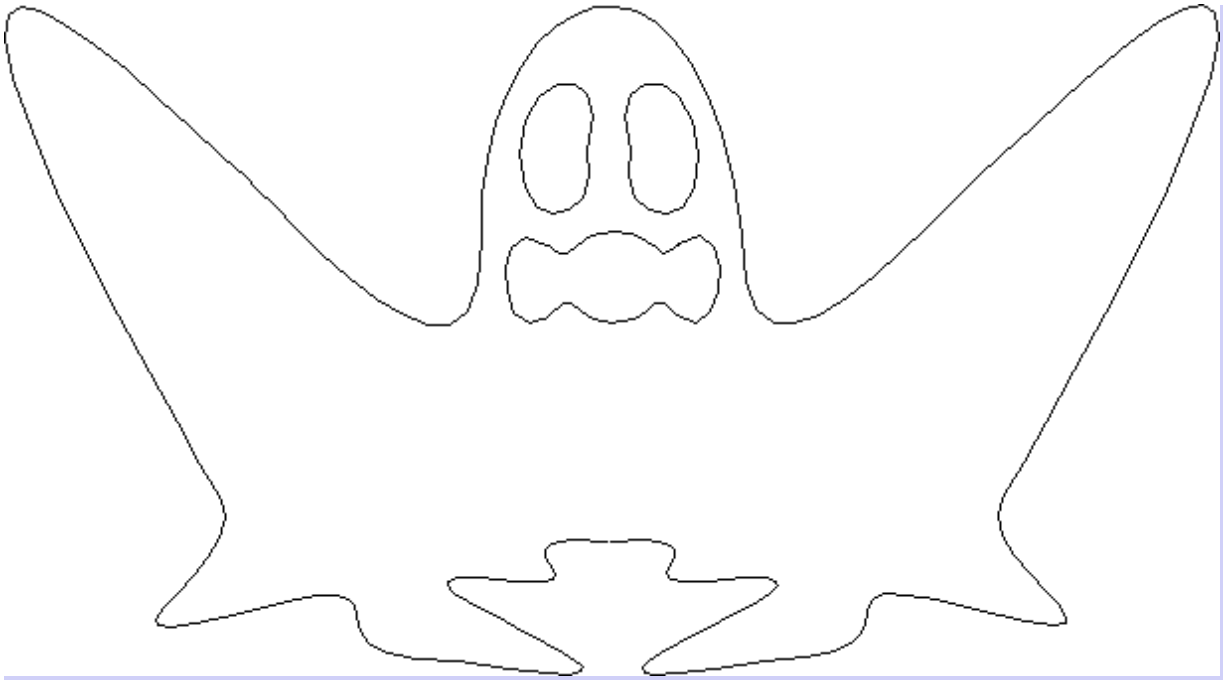


Figura 23 - Fantasma (Bézier)

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 máquina virtual de Sun (Sun VM) para poder ejecutar este applet de Java.

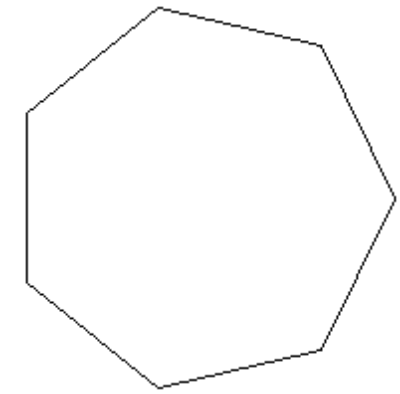
Ejercicios

Enlace al paquete *de este capítulo.*

1. Escriba un programa que muestre cualquier polígono regular, como se ha descrito en este capítulo. Intente con los siguientes

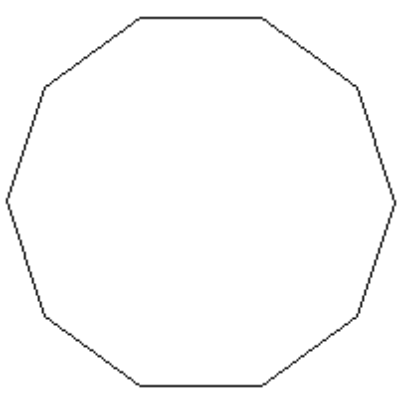
parámetros:

a. N = 7



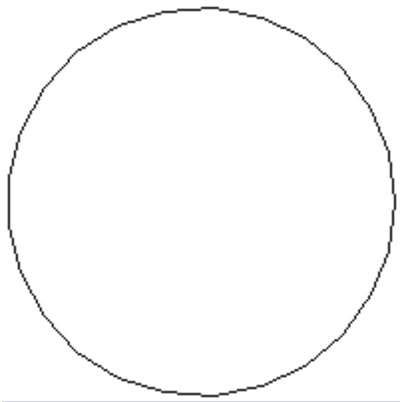
Ejercicio 1a

b. N = 10



Ejercicio 1b

c. N = 25



Ejercicio 1c

Elija valores para la resolución y el radio que convengan.

Como una extensión al algoritmo, podemos agregar un ángulo inicial, $\acute{a}ngulo_i$. 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. Escriba 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
```

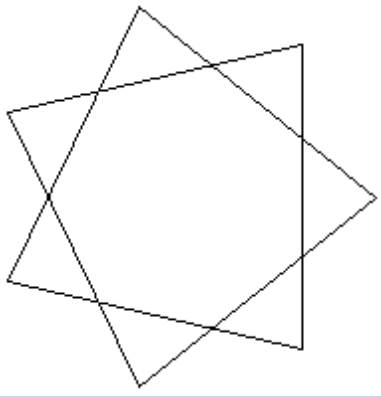
```
            4. vértices[i].x ← centro.x + radio * cos(
               (i-1)*alfa + ángulo_i )
            5. vértices[i].y ← centro.y + radio * sen(
               (i-1)*alfa + ángulo_i )


  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.
```

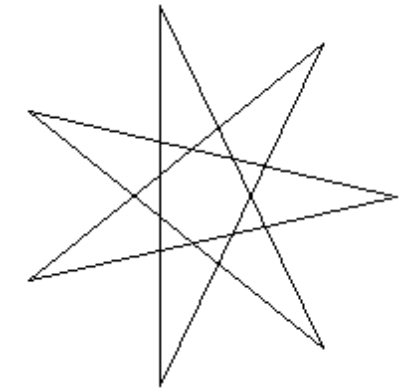
Pruebe con los siguientes valores:

a. N = 7, salto = 2



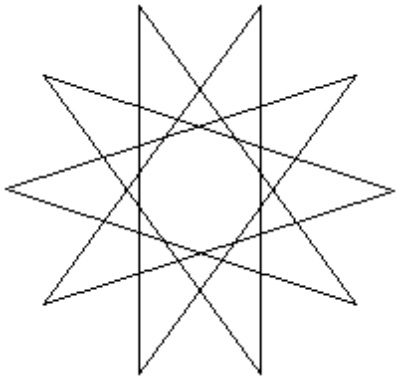
Ejercicio 2a

b. N = 7, salto = 3



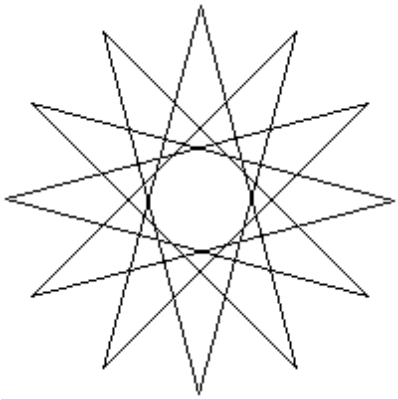
Ejercicio 2b

c. N = 10, salto = 4



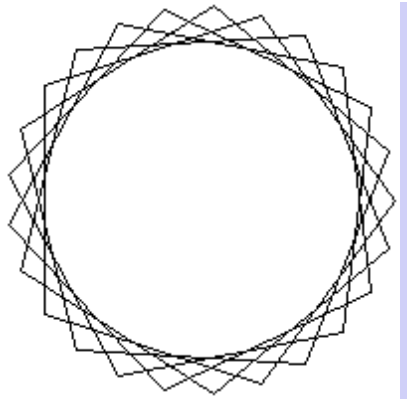
Ejercicio 2c

d. N = 12, salto = 5



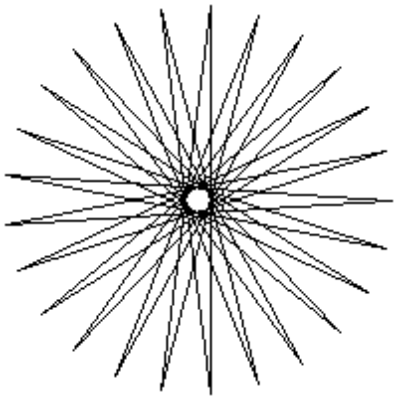
Ejercicio 2d

e. N = 25, salto = 5



Ejercicio 2e

f. N = 25, salto = 12

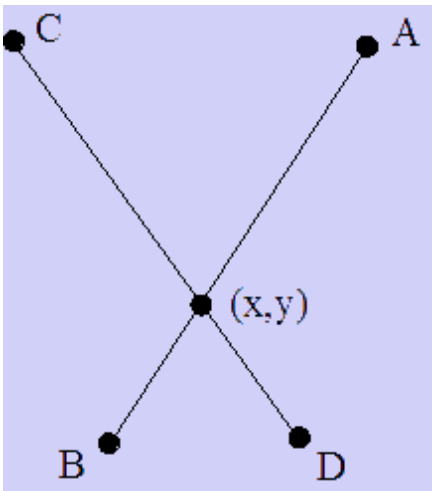


Ejercicio 2f

3. Muestre 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 crucen 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.



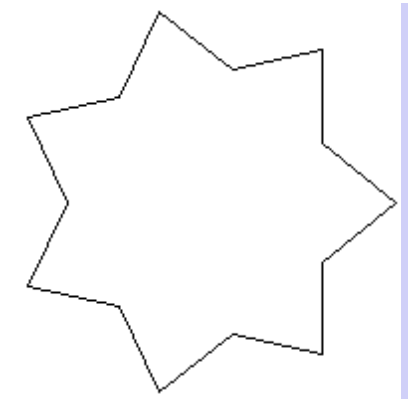
Ejercicio 3 - Segmentos cortantes

Simplificando, obtenemos que,

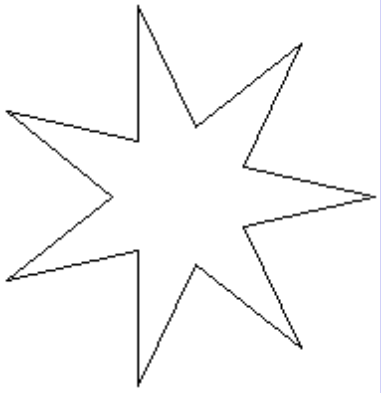
$x = (m*A_x - A_y - n*C_x + C_y) / (m-n),$
$y = m*(x - A_x) + A_y,$

donde,

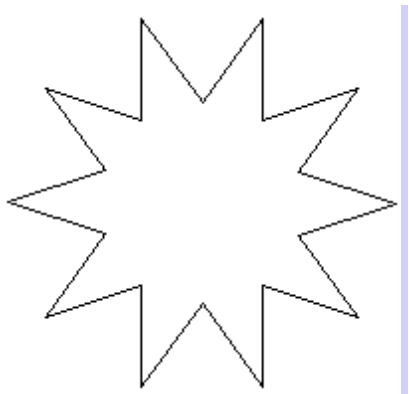
$m = (B_y - A_y) / (B_x - A_x),$ y
$n = (D_y - C_y) / (D_x - C_x)$



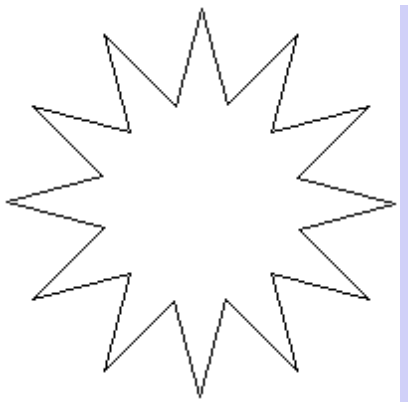
Ejercicio 3a



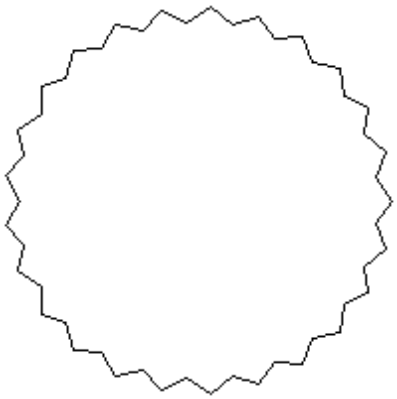
Ejercicio 3b



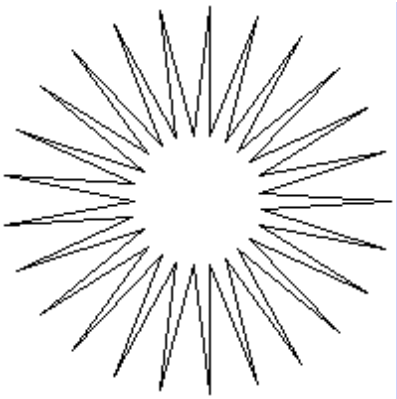
Ejercicio 3c



Ejercicio 3d



Ejercicio 3e



Ejercicio 3f

Algunas sugerencias para la implementación son:

1. Calcule los vértices de la estrella, como se ha explicado anteriormente, en el ejercicio #2. Luego, calcule 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. Cree 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:

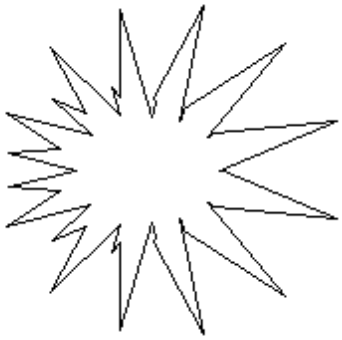
```
        4. vértices[i].x ← centro.x + radio((i-1)*alfa +
ángulo_i) * cos((i-1)*alfa + ángulo_i)
        5. vértices[i].y ← centro.y + radio((i-1)*alfa +
ángulo_i) * sen((i-1)*alfa + ángulo_i)
```

Pruebe con los siguientes valores y funciones:

a. N = 25, salto = 11,

```
Real radio( t )
 1. a ← 12
 2. b ← 30
```

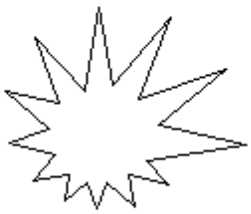
```
  3. x ← a*cos( t ) + b*sen( a*t/2 )
  4. y ← a*cos( t ) - b*sen( a*t/2 )
  5. resultado ← √( (x²+y²) / ((a+b)² + (a-b)²) )
  6. Terminar( resultado )
```



Ejercicio 4a

b. N = 12, salto = 5,

```
Real radio( t )
  1. a ← 0,1
  2. b ← 1
  3. x ← 2*a*cos( t ) + b*cos( a*t )
  4. y ← 2*a*sen( t ) - b*cos( a*t )
  5. resultado ← √( (x+y) / ((2*a+b)² + (2*a-b)²) )
  6. Terminar( resultado )
```

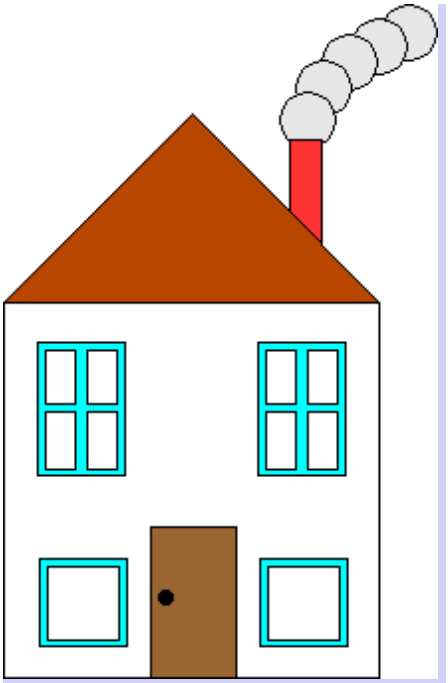


Ejercicio 4b

5. Basándose en la figura 17, dibuje 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..

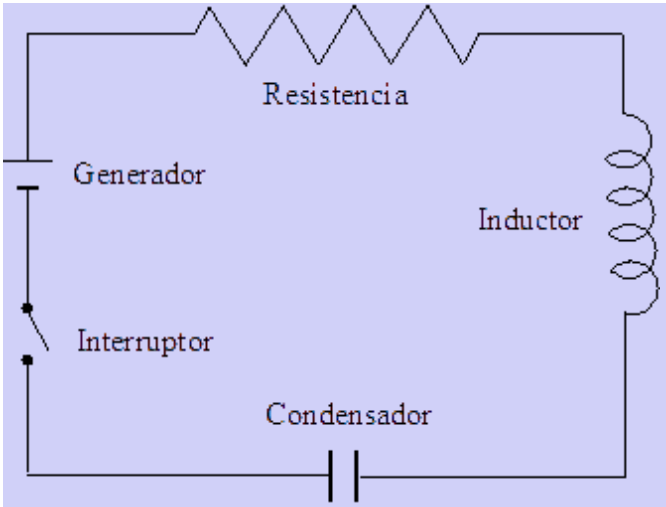
6. Usando la técnica de composición, dibuje una casa. Puede basarse en la siguiente ilustración.



Ejercicio 6

7. Cree 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, muestre 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. Cree 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

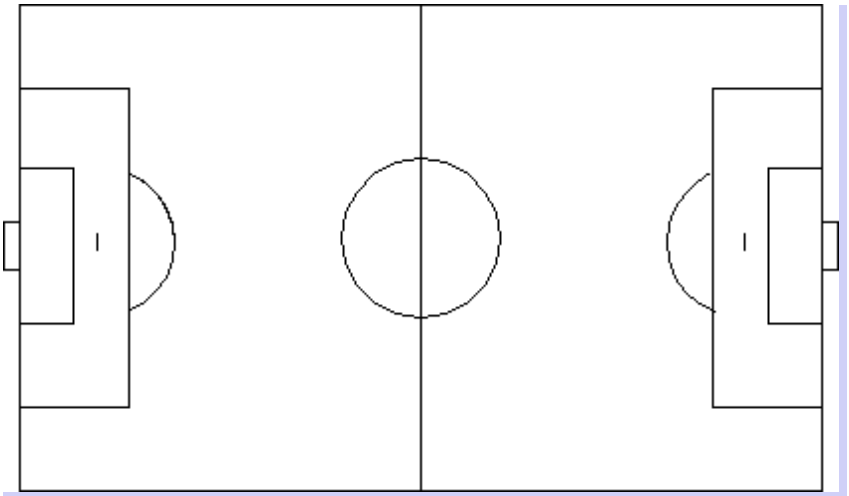
podría crear un circuito con dos generadores eléctricos, cuatro condensadores, tres resistencias, dos inductores, y un interruptor. La complejidad del programa es responsabilidad del programador; se puede crear un programa altamente adaptable a cada tipo de circuito o un programa muy simple que agrega un componente detrás de otro, secuencialmente.



Ejercicio 8

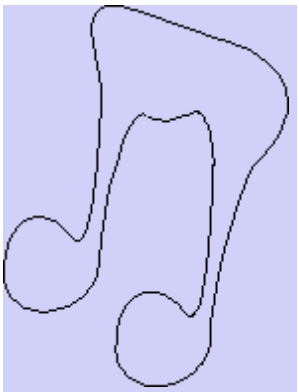
Como una sugerencia, cree funciones para poder dibujar cada componente de forma general. El programa principal invocará cada función para dibujar el componente eléctrico según los datos que describen el circuito. La descripción del circuito puede ser simple, como una cadena de caracteres. Por ejemplo,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

9. Diseñe un campo de fútbol. Se puede basar en el siguiente diagrama.



Ejercicio 9

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



Ejercicio 10

Capítulo 6 - Modelado en 2D

Hasta este momento se ha visto varios ejemplos de programación gráfica implementando diferentes técnicas. A partir de ahora nos centraremos en generar nuestra imagen como un conjunto de objetos visuales. Crearemos un modelo para representar nuestra escena describiendo los objetos que la componen. Representaremos cada objeto en términos de su geometría y topología. La necesidad de colocar los objetos en la escena nos fuerza a describirlos según su posición. Nos basaremos en un sistema de coordenadas, como el plano cartesiano, para situar cada uno de estos objetos.

Para crear y manipular modelos para poder crear las escenas que compondrán nuestra imagen, necesitamos describir nuestros modelos y sus operaciones numérica y geométricamente. Esto supone usar las matemáticas para poder desarrollar las imágenes que nos interesan generar.

Conceptos

Para poder entender las operaciones y algoritmos que usaremos para modelar nuestros objetos y escenas, debemos comprender algunos conceptos matemáticos fundamentales.

Vamos a definir brevemente los conceptos acerca de tres tipos básicos de entidades para describir nuestros objetos y que nos proporcionarán ciertas propiedades importantes:

Escalares

Básicamente, se trata de números reales que tienen dos operaciones importantes: la suma y la multiplicación. También tienen las siguientes propiedades:

- Conmutativa:

```
a + b = b + a
a * b = b * a
```

- Asociativa:

```
a + (b + c) = (a + b) + c
a * (b * c) = (a * b) * c
```

- Distributiva:

```
a * (b + c) = (a * b) + (a * c)
```

Existen dos escalares especiales: el elemento neutro 0 para la suma y el 1 para la multiplicación. Esto es,

$$a + 0 = 0 + a = a$$
$$a * 1 = 1 * a = a$$

Además, existen las inversas de la suma, descrita como -a, y de la multiplicación, descrita como a^{-1}. Esto es,

$$a + (-a) = 0$$
$$a * a^{-1} = 1$$

Puntos

Un punto no es nada más que una ubicación en un espacio sin siquiera tener tamaño. Los puntos existen irrelevantemente de un sistema de referencia o de coordenadas. Obviamente, sería inconveniente e impráctico describir cada punto existente sin usar un sistema de coordenadas.

Vectores

Geométricamente, un vector es un segmento orientado que tiene dirección, sentido (u orientación), y magnitud, representada como la longitud del segmento. Un vector no tiene posición, por lo que dos

vectores son idénticos si tienen la misma dirección, orientación, y magnitud. Al igual que ocurre con los puntos, necesitaremos un sistema de coordenadas para referirnos a cualesquier vectores.

Nota: Sugerimos consultar el Apéndice 2 para profundizar en el tema de los vectores.

Espacios

Nos interesa hablar de ciertos espacios concretos, como son:

- El espacio vectorial, o a veces llamado el espacio vectorial lineal, se compone de escalares y de vectores. Se definen varias operaciones con estas entidades y entre sí.

  Nota: Consulten el Apéndice 2 para más información acerca de estas operaciones.
- El espacio afín amplía el espacio vectorial agregando el concepto de puntos. Con estas entidades, podemos crear un sistema de coordenadas, descrito como una base de vectores y el punto de origen.
- El espacio euclidiano agrega al espacio afín la noción de distancia o la medida de tamaño. La nueva operación principal es el llamado producto interno que resulta en un escalar a partir de dos vectores. Por esta razón, se suele llamar producto escalar a esta operación.

Líneas

Una línea recta es un conjunto infinito de puntos que comparte una misma pendiente o dirección. Podemos describir cualquier punto en una línea con las entidades descritas anteriormente; esto es,

$$P(t) = P_0 + t\ \mathbf{v}$$

donde P_0 es un punto arbitrario y conocido, *t* es un escalar, y **v** es un vector. P(t) describirá cualquier punto en la línea según el parámetro *t*.

Al ajustar un sistema de coordenadas, la fórmula de una línea recta, en dos dimensiones, se convierte en la siguiente:

$$(x(t),y(t)) = (x_0,y_0) + t\ (v_x,v_y)$$

Se puede convertir lo anterior en el siguiente sistema de ecuaciones:

$$x(t) = x_0 + t\ v_x$$
$$y(t) = y_0 + t\ v_y$$

Despejando el parámetro t de este sistema de ecuaciones, obtenemos la siguiente fórmula más familiar:

$$(y(x)-y_0) = m\ (x-x_0)$$

donde *m* es la pendiente que es igual a v_y/v_x. Esta ecuación termina siendo el siguiente polinomio,

$$y(x) = m\ x + b$$

Convexidad

Un polígono es convexo si cada segmento que une cada dos de sus vértices existe completamente en el interior de tal polígono. También podemos decir que un polígono es convexo si el ángulo formado por cada pareja contigua de aristas queda comprendida entre 0° y 180° (ambos incluidos).

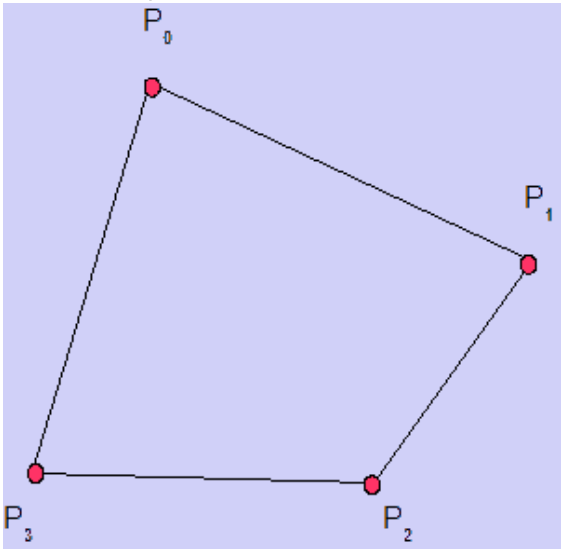


Figura 1 - Convexo

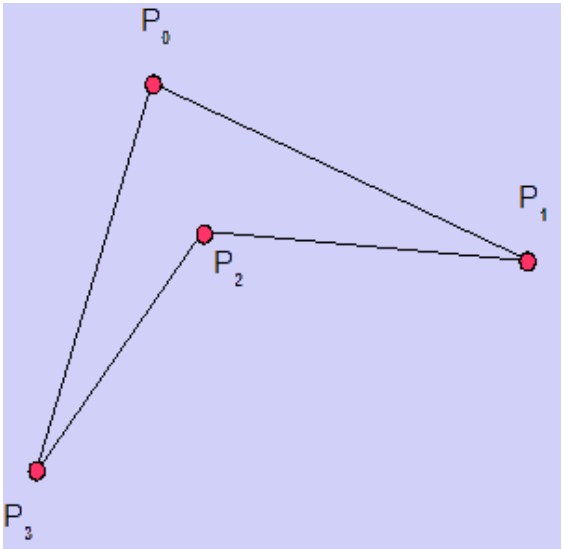


Figura 2 - Cóncavo

Ahora podemos continuar con otros conceptos que necesitamos para crear y controlar los modelos de los objetos gráficos. Los objetos que vamos a tratar por el momento serán compuestos por uno o varios polígonos convexos. Al manejar polígonos, podemos describirlos a través de un conjunto de vértices y aristas. Trabajar con polígonos, vértices, y aristas es más simple que manipular objetos más complejos basados en fórmulas y funciones matemáticas complicadas. Si usáramos objetos complejos, se nos complicaría bastante la generación de la imagen. En otros capítulos, veremos otras técnicas avanzadas para crear modelos más complicados pero más realistas. Por el momento, veremos las técnicas necesarias para generar y visualizar objetos basados en modelos que aproximen aquellos objetos más complicados.

Sistema de Coordenadas

Hemos visto los objetos abstractos: escalares, puntos, y vectores. Ahora asociaremos estos objetos a un sistema de coordenadas como referencia.

El sistema de coordenadas o sistema de referencia es un conjunto de N escalares para referirse unívocamente a cada punto en un espacio de N dimensiones. Por ejemplo, en dos dimensiones (2D), cada punto es descrito por dos escalares que representan dos coordenadas. El típico sistema de coordenadas que usamos es el llamado sistema de coordenadas cartesianas o rectangulares. Este sistema de coordenadas se basa en anchura y altura - las mismas propiedades de un rectángulo. También es usual usar el sistema de coordenadas polares, que se basa en un radio y en un ángulo, para describir cualquier punto.

Por el momento nos ceñiremos a usar un sistema de coordenadas rectangulares. Describimos este sistema a través de una base - conjunto de vectores linealmente independientes. Por ejemplo, podemos describir un vector cualquiera, **q**, de esta manera:

$$\mathbf{q} = a\,\mathbf{u} + b\,\mathbf{v}$$

donde *a* y *b* son escalares y componentes de **q**, mientras que **u** y **v** son vectores que forman una base.

Una base involucra vectores, pero como los vectores no tienen posición, necesitamos un punto de referencia para nuestro sistema de coordenadas. Por ello, creamos un marco de referencia en espacios afines. Un marco de referencia se compone de la base que describe un sistema de coordenadas y un punto de referencia que describe el origen. De esta forma, fijamos los vectores de la base en un solo punto. Ahora podemos describir cualquier vector en un marco de la misma forma que en el espacio vectorial:

$$\mathbf{q} = a\,\mathbf{u} + b\,\mathbf{v}$$

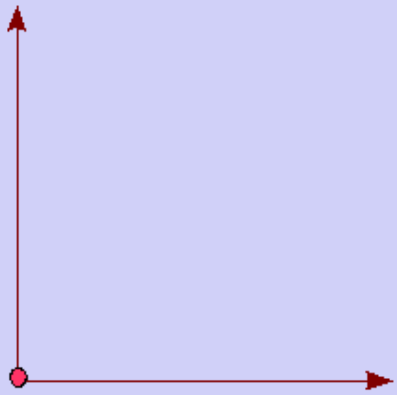


Figura 3 - Marco de referencia

También podemos describir únicamente cualquier punto, P, en un marco, de esta manera:

$$P = O + a\,\mathbf{u} + b\,\mathbf{v}$$

donde O es el punto de origen.

Cambio de Coordenadas

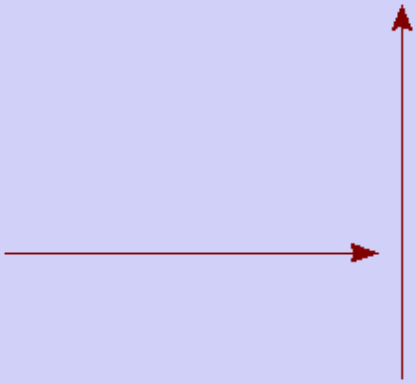


Figura 4 - Sistema de coordenadas con los vectores mudados

Frecuentemente, querremos representar un punto o un vector en una base a su equivalente en otra base. Esto supone cambiar de un sistema de coordenadas a otro. Digamos que tenemos dos bases: $\{\mathbf{u}_1,\mathbf{u}_2\}$ y $\{\mathbf{v}_1,\mathbf{v}_2\}$. Cada vector en la primera base se puede representar en términos de la segunda base, y viceversa. Esta representación incluye

componentes escalares resultando en el siguiente sistema de definiciones,

$$\mathbf{u_1} = a_{11}\,\mathbf{v_1} + a_{12}\,\mathbf{v_2}$$
$$\mathbf{u_2} = a_{21}\,\mathbf{v_1} + a_{22}\,\mathbf{v_2}$$

Describimos el conjunto de los componentes escalares como una matriz; esto es,

$$\mathbf{M} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$

Usando matrices, obtenemos la siguiente expresión,

$$\begin{pmatrix} \mathbf{u_1} \\ \mathbf{u_2} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} \mathbf{v_1} \\ \mathbf{v_2} \end{pmatrix}$$

Para poder cambiar la representación de $\{\mathbf{v_1},\mathbf{v_2}\}$ a $\{\mathbf{u_1},\mathbf{u_2}\}$ debemos calcular la inversa de la matriz, **M**, de los componentes escalares. En general, representamos cualquier vector, **v**, en la base $\{\mathbf{v_1},\mathbf{v_2}\}$ de esta manera, en forma matricial,

$$\mathbf{v} = \begin{pmatrix} v_x & v_y \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$$

Para representar **v** en la base de $\{\mathbf{u_1},\mathbf{u_2}\}$, acabaremos invirtiendo la matriz, **M**. Si **u** es la nueva representación del mismo vector, **v**, entonces la expresión es la siguiente,

$$\mathbf{u} = \begin{pmatrix} u_x & u_y \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$$

Para llevar a cabo la conversión, queremos averiguar los valores de (u_x, u_y). Realizamos los siguientes pasos para ello,

$$\begin{pmatrix} u_x & u_y \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} v_x & v_y \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$$

Sustituimos la base {$\mathbf{u_1}$,$\mathbf{u_2}$} por su definición,

$$\begin{pmatrix} u_x & u_y \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} v_x & v_y \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$$

Como la matriz de la base {$\mathbf{v_1}$,$\mathbf{v_2}$} multiplica ambos lados de la igualación, la eliminamos. Ahora tenemos lo siguiente:

$$\begin{pmatrix} u_x & u_y \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} = \begin{pmatrix} v_x & v_y \end{pmatrix}$$

Como conocemos la matriz, **M**, y el vector, **v**, necesitamos despejar la matriz del lado izquierdo de la igualdad, para determinar los valores desconocidos que representa el vector, **u**. Esto es,

$$\mathbf{u\ M} = \mathbf{v}$$
$$\mathbf{u\ M\ M}^{-1} = \mathbf{v\ M}^{-1}$$
$$\mathbf{u\ I} = \mathbf{v\ M}^{-1}$$
$$\mathbf{u} = \mathbf{v\ M}^{-1}$$

La ecuación final es la siguiente, que se basa en calcular la inversa de la matriz, **M**,

$$\mathbf{u} = \mathbf{v} \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}^{-1}$$

Este cambio de bases no varía el origen y por tanto podemos usarlo para describir rotaciones y cambios de escala de una base en términos de otra base. Sin embargo, para realizar una traslación del origen o un cambio de marco, no se puede representar de esta forma. Antes de ver este cambio de marco, veamos un ejemplo de un cambio de coordenadas.

Nota: Aconsejamos ver el Apéndice 3 para una introducción a las matrices.

Ejemplo

Tenemos el vector, **v** = (-1, 3), en la base {**i**,**j**}, que se representa así:

$$\mathbf{v} = -1\ \mathbf{i} + 3\ \mathbf{j}$$

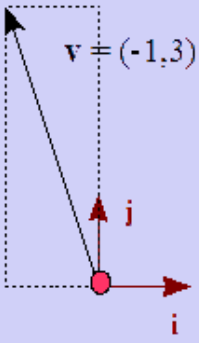


Figura 5 - Definición de v

Ahora queremos representar **v** en la base $\{\mathbf{u_1},\mathbf{u_2}\}$ cuya representación es la siguiente, en términos de {**i**,**j**}:

$$\mathbf{u_1} = -1\ \mathbf{i} + 1\ \mathbf{j}$$
$$\mathbf{u_2} = 2\ \mathbf{i} + 2\ \mathbf{j}$$

Por lo tanto, la matriz de los componentes de los vectores es la siguiente:

$$\mathbf{M} = \begin{pmatrix} -1 & 1 \\ 2 & 2 \end{pmatrix}$$

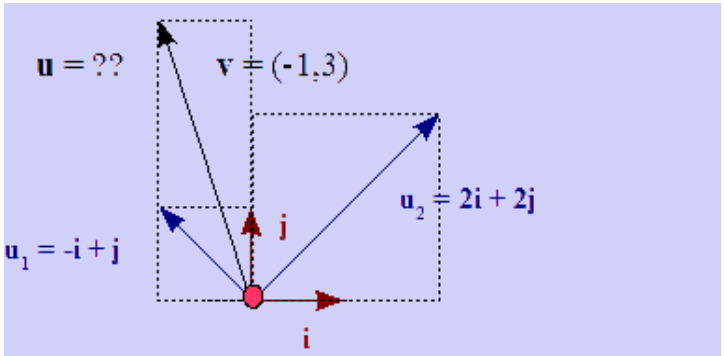


Figura 6 - Definiciones de v y de {u1,u2}

Para realizar el cambio de coordenadas, necesitamos calcular la inversa de **M**, que es,

$$\mathbf{M}^{-1} = \begin{pmatrix} -1 & 1 \\ 2 & 2 \end{pmatrix}^{-1} = A$$

$$A = \begin{pmatrix} -0{,}50 & 0{,}25 \\ 0{,}50 & 0{,}25 \end{pmatrix}$$

En la nueva representación, llamemos a **v** el vector, **u**, cuya representación es la siguiente:

$$\mathbf{u} = \mathbf{v}\ \mathbf{A}$$

$$\mathbf{u} = \begin{pmatrix} -1 & 3 \end{pmatrix} \begin{pmatrix} -0{,}50 & 0{,}25 \\ 0{,}50 & 0{,}25 \end{pmatrix}$$

v = (-1, 3) en la base {**i**,**j**} se representa como **u** = (2, 0,50) en la base $\{\mathbf{u_1},\mathbf{u_2}\}$.

Coordenadas Homogéneas

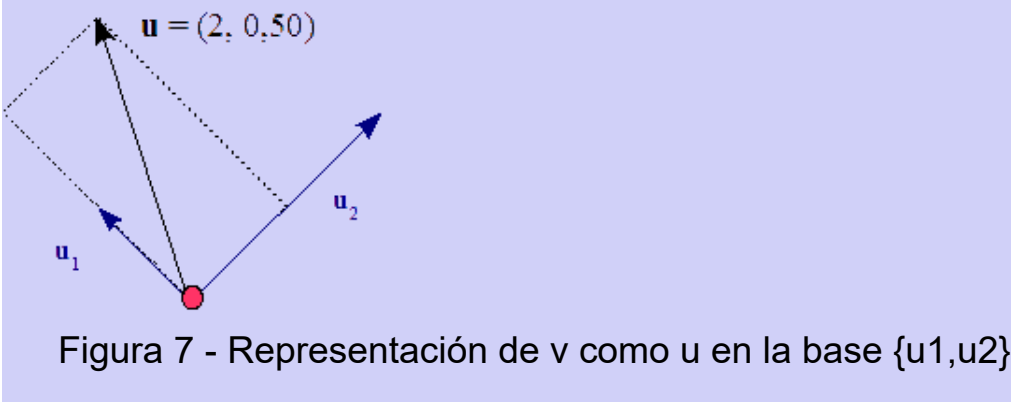


Figura 7 - Representación de v como u en la base {u1,u2}

Uno de los problemas con que nos enfrentamos es la posible confusión entre un punto y un vector, a la hora de representarlos. Por ejemplo, P = (3 2) y **v** = (-2 5) comparten la misma representación matricial. Además, no se puede representar un cambio de marco con multiplicaciones matriciales. Para evitar estas dificultades, usaremos coordenadas homogéneas las cuales agregan un elemento o dimensión más al que tenemos, para representar puntos y vectores.

En el marco descrito por ($\mathbf{u_1}$, $\mathbf{u_2}$, P_0), cualquier punto, P, puede ser representado como,

$$P = a\ \mathbf{u_1} + b\ \mathbf{u_2} + P_0$$

Podemos expresar lo anterior con matrices, resultando en lo siguiente:

$$P = \begin{pmatrix} a & b & 1 \end{pmatrix} \begin{pmatrix} \mathbf{u_1} \\ \mathbf{u_2} \\ P_0 \end{pmatrix}$$

Asimismo, podemos expresar cualquier vector del mismo marco de la siguiente manera:

$$\mathbf{v} = c\ \mathbf{u_1} + d\ \mathbf{u_2}$$

La expresión en forma matricial es la siguiente:

$$\mathbf{v} = \begin{pmatrix} c & d & 0 \end{pmatrix} \begin{pmatrix} \mathbf{u_1} \\ \mathbf{u_2} \\ P_0 \end{pmatrix}$$

Podemos expresar este vector sencillamente como:

$$\mathbf{v} = (c \; d \; 0)$$

Resumiendo, con las coordenadas homogéneas,

- Podemos representar y manipular puntos y vectores de la misma manera.
- Cualquier punto en 2D se representará así, P = (x, y, 1).
- Cualquier vector en 2D se representará así, $\mathbf{v} = (v_x, v_y, 0)$.

Existe una relación lineal entre un punto en 2D y su representación en coordenadas homogéneas. Al extender un punto en 2D a uno en 3D, éste se convierte en una línea recta de la forma, P = (tx, ty, tw). Por lo tanto, tenemos un conjunto de coordenadas equivalentes, como es (2, 3, 1), (4, 6, 2), (20, 30, 10), etcétera. Sin embargo, como nos interesa mantener el tercer componente como 1, debemos homogeneizar el trío. Esto se hace dividiendo el tercer componente, w, entre todos. Por ejemplo, P = (-15, 10, 5) pasa a ser P = (-3, 2, 1).

Cambio de Marcos de Referencia

Usando coordenadas homogéneas, podemos realizar un cambio de marcos de referencia entre $\{\mathbf{v_1}, \mathbf{v_2}, P_0\}$ y $\{\mathbf{u_1}, \mathbf{u_2}, Q_0\}$. Como hicimos con el cambio de coordenadas, expresaremos un marco en términos del otro. Esto es,

$$\mathbf{u_1} = a_{11} \mathbf{v_1} + a_{12} \mathbf{v_2}$$
$$\mathbf{u_2} = a_{21} \mathbf{v_1} + a_{22} \mathbf{v_2}$$
$$Q_0 = a_{31} \mathbf{v_1} + a_{32} \mathbf{v_2} + P_0$$

En forma matricial, obtenemos la siguiente fórmula,

$$\begin{pmatrix} \mathbf{u_1} \\ \mathbf{u_2} \\ Q_0 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & 1 \end{pmatrix} \begin{pmatrix} \mathbf{v_1} \\ \mathbf{v_2} \\ P_0 \end{pmatrix}$$

La matriz de componentes escalares es la llamada *matriz de representación* del cambio de marcos de referencia. Se define de esta manera:

$$\mathbf{M} = \begin{pmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & 1 \end{pmatrix}$$

Podemos cambiar la representación de un punto o de un vector al tener diferentes marcos de referencia. Si **v** es un vector descrito en el marco de $\{\mathbf{v_1}, \mathbf{v_2}, P_0\}$,

$$\mathbf{v} = \begin{pmatrix} v_x & v_y & 0 \end{pmatrix} \begin{pmatrix} \mathbf{v_1} \\ \mathbf{v_2} \\ P_0 \end{pmatrix}$$

su representación, **u**, en el marco de $\{\mathbf{u_1}, \mathbf{u_2}, Q_0\}$ es,

$$\mathbf{u} = \begin{pmatrix} u_x & u_y & 0 \end{pmatrix} \begin{pmatrix} \mathbf{u_1} \\ \mathbf{u_2} \\ Q_0 \end{pmatrix}$$

Para calcular los escalares de $\mathbf{u} = (u_x, u_y, 0)$, debemos calcular la inversa de la matriz, **M**. Esto es,

$$\mathbf{u} = \mathbf{v} \begin{pmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & 1 \end{pmatrix}^{-1}$$

Ejemplos de Cambio de Marcos

Veamos dos ejemplos de cambio de marcos de referencia.

1. Volvamos al ejemplo anterior:
   Tenemos el vector, $\mathbf{v} = (-1, 3, 0)$, en el marco $\{\mathbf{i}, \mathbf{j}, (0,0)\}$, que se representa así:

$$\mathbf{v} = -1\ \mathbf{i} + 3\ \mathbf{j}$$

Ahora queremos representar **v** en el marco {$\mathbf{u_1}$,$\mathbf{u_2}$, Q_0} cuya representación es la siguiente, en términos de {**i**,**j**, (0,0)}:

$$\mathbf{u_1} = -1\ \mathbf{i} + 1\ \mathbf{j}$$
$$\mathbf{u_2} = 2\ \mathbf{i} + 2\ \mathbf{j}$$
$$Q_0 = 2\ \mathbf{i} - 1\ \mathbf{j} + (0,0)$$

La matriz de la representación es la siguiente,

$$\mathbf{M} = \begin{pmatrix} -1 & 1 & 0 \\ 2 & 2 & 0 \\ 2 & -1 & 1 \end{pmatrix}$$

La inversa de la matriz, **M**, es la siguiente matriz, **A**,

$$\mathbf{M^{-1}} = \begin{pmatrix} -1 & 1 & 0 \\ 2 & 2 & 0 \\ 2 & -1 & 1 \end{pmatrix}^{-1} = \mathbf{A}$$

$$\mathbf{A} = \begin{pmatrix} -0{,}50 & 0{,}25 & 0 \\ 0{,}50 & 0{,}25 & 0 \\ 1{,}50 & -0{,}25 & 1 \end{pmatrix}$$

El vector, **v**, representado en el nuevo marco se llamará **u**, que se define como,

$$\mathbf{u} = \begin{pmatrix} -1 & 3 & 0 \end{pmatrix} \begin{pmatrix} -0{,}50 & 0{,}25 & 0 \\ 0{,}50 & 0{,}25 & 0 \\ 1{,}50 & -0{,}25 & 1 \end{pmatrix} = \begin{pmatrix} 2 & 0{,}50 & 0 \end{pmatrix}$$

Como podemos observar, obtenemos el mismo resultado que en el ejemplo anterior, cuando hicimos un cambio de coordenadas. Esto es porque los vectores no tienen posición y por tanto el cambio del punto de origen no afecta a los vectores ni contribuye a su representación.

2. Veamos lo que sucede al representar el punto, P = (1, -1) en el marco {**i**,**j**, (0,0)}:

$$P = 1\ \mathbf{i} - 1\ \mathbf{j} + (0,0) = \mathbf{i} - \mathbf{j}$$

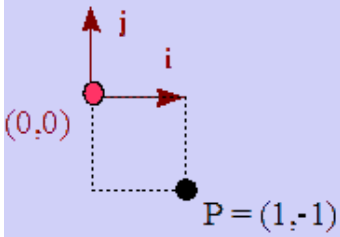


Figura 8 - Definición de P en el marco {i,j, (0,0)}

Ahora queremos representar el punto, P, en el marco {$\mathbf{u_1}$,$\mathbf{u_2}$, (2,-1)}. La matriz de representación de este marco es la siguiente:

$$\mathbf{M} = \begin{pmatrix} -1 & 1 & 0 \\ 2 & 2 & 0 \\ 2 & -1 & 1 \end{pmatrix}$$

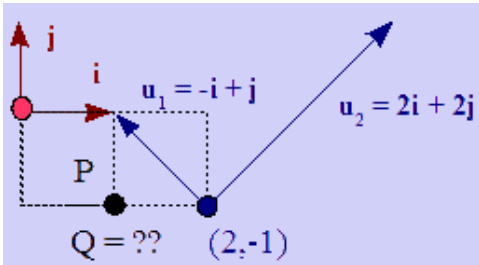


Figura 9 - Definiciones de P y el marco {u1,u2, (2,-1)} en el marco {i,j, (0,0)}

La nueva representación del punto, P, se calcula de la siguiente manera:

$$Q = P\,\mathbf{M}^{-1}$$

$$Q = \begin{pmatrix} 1 & -1 & 1 \end{pmatrix} \begin{pmatrix} -0{,}50 & 0{,}25 & 0 \\ 0{,}50 & 0{,}25 & 0 \\ 1{,}50 & -0{,}25 & 1 \end{pmatrix} = \begin{pmatrix} 0{,}50 & -0{,}25 & 1 \end{pmatrix}$$

Se trata del mismo punto en el espacio euclidiano, pero según el marco de referencia, se puede representar de dos formas diferentes: (1, -1) ó (0,50, -0,25).

Transformaciones Afines

Una transformación es una función que acepta un punto y lo convierte - siguiendo alguna lógica o algoritmo - en otro punto. Asimismo se puede

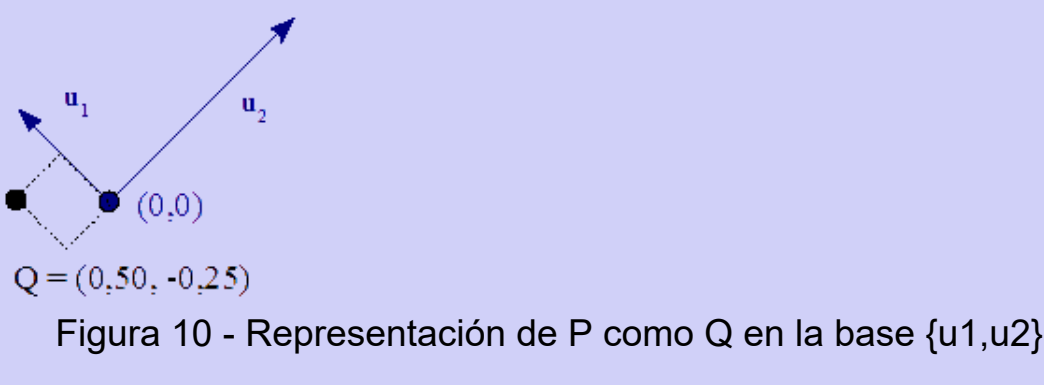


Figura 10 - Representación de P como Q en la base {u1,u2}

decir de una transformación de un vector a otro vector. Si usamos coordenadas homogéneas, podemos representar cualquier transformación usando la misma función para aceptar tanto puntos como vectores. El problema es que esta definición es demasiado general para nuestros propósitos. Como vamos a estar manejando objetos geométricos más complicados que puntos y vectores, sería impráctico aplicar la misma transformación a cada punto y vector que compone nuestros objetos en nuestra escena y en nuestra imagen.

Necesitamos limitar las transformaciones que queremos usar. La restricción más importante es la linealidad. Sin entrar en mucho detalle, el uso de transformaciones lineales supone que una línea se convierte en una línea. De esta manera, podemos transformar los puntos finales que describen un segmento para así transformar tal segmento, sin tener que transformar todos los puntos que comprenden tal segmento. Usando coordenadas homogéneas, tales transformaciones implicarán una multiplicación matricial. Podemos representar tal transformación entre *a* y *b*, que pueden representar tanto puntos como vectores, de esta manera,

```
a = b M
```

Como podemos ver, esta fórmula es idéntica a la que usamos para el cambio de marco de referencia. Por lo tanto, una transformación lineal corresponde a un cambio de marco. Dicho esto, declaramos que una transformación es a) un cambio de representación de los objetos de un marco de referencia a otro o b) un cambo de las figuras de los objetos dentro del mismo marco de referencia.

Volviendo a la matriz en coordenadas homogéneas,

$$\mathbf{M} = \begin{pmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & 1 \end{pmatrix}$$

vemos que sólo tenemos 6 elementos con los que trabajar de los 9 que existen en una matriz de 3x3. Técnicamente, decimos que esta matriz tiene 6 grados de libertad. Sin embargo, al aplicar este tipo de matrices para transformar un punto o un vector, podemos deducir dos tipos de matrices diferentes. Como un vector se define así,

$$\mathbf{v} = \begin{pmatrix} v_x & v_y & 0 \end{pmatrix}$$

la transformación acaba siendo la siguiente matriz,

$$\mathbf{M}_{\mathbf{vector}} = \begin{pmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Como la coordenada homogénea de un vector es cero, los elementos a_{31} y a_{32} de la matriz no influyen en el resultado de la multiplicación.

Para transformar un punto, descrito como,

$$P = \begin{pmatrix} P_x & P_y & 1 \end{pmatrix}$$

la transformación es la siguiente matriz,

$$\mathbf{M}_{\mathbf{punto}} = \begin{pmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & 1 \end{pmatrix}$$

Por lo tanto, existen 4 grados de libertad para una transformación de vectores, pero usamos los 6 grados para las transformaciones afines con puntos.

Operaciones

Existen cuatro transformaciones afines: traslación, cambio de escala, rotación, y sesgado o transvección. Usando coordenadas homogéneas,

cada transformación se puede representar como una matriz de 3x3 de la forma,

$$\mathbf{M} = \begin{pmatrix} a_{11} & a_{12} & 0 \\ a_{21} & a_{22} & 0 \\ a_{31} & a_{32} & 1 \end{pmatrix}$$

Traslación

Se trata de una operación que desplaza un punto una distancia fija en una dirección y sentido concretos. El parámetro que se necesita es simplemente un vector a modo de desplazamiento: $\mathbf{d} = (d_x, d_y)$,

$$P' = P + \mathbf{d}$$

donde,

$$P = (P_x \quad P_y \quad 1)$$
$$P' = (P'_x \quad P'_y \quad 1)$$
$$\mathbf{d} = (d_x \quad d_y \quad 0)$$

Podemos reescribir estas ecuaciones por componentes,

$$P'_x = P_x + d_x$$
$$P'_y = P_y + d_y$$

Como vemos, una traslación trata de una suma. Esto va en contra de la multiplicación matricial que

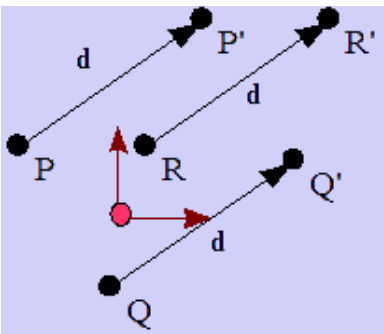


Figura 11 - Traslación de varios puntos por un vector d

hemos estado aplicando hasta estos momentos. Sin embargo, no tenemos un grave problema al usar coordenadas homogéneas,

$$P' = P\ \mathbf{T}$$

donde,

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ d_x & d_y & 1 \end{pmatrix}$$

Analizando esta matriz, vemos que tiene 2 grados de libertad. Por ello, solemos describir una traslación como una función que acepta dos parámetros: $\mathbf{T}(d_x,d_y)$.

Como no podemos conseguir el mismo resultado con una matriz de 2x2, se usa una matriz de 3x3 y coordenadas homogéneas como una forma ingeniosa de convertir una suma a una multiplicación. Esto nos permitirá "encadenar" las transformaciones como multiplicaciones entre matrices. Con una suma de por medio, no podrímos hacer tal "encadenación", ya que las multiplicaciones se deben realizar antes de las sumas.

Podríamos calcular la inversa de la matriz de traslación con el algoritmo general. Sin embargo, para la traslación, existe una solución sencilla basada en la geometría. La inversa de una traslación es sencillamente otra traslación en el sentido contrario. Esto es,

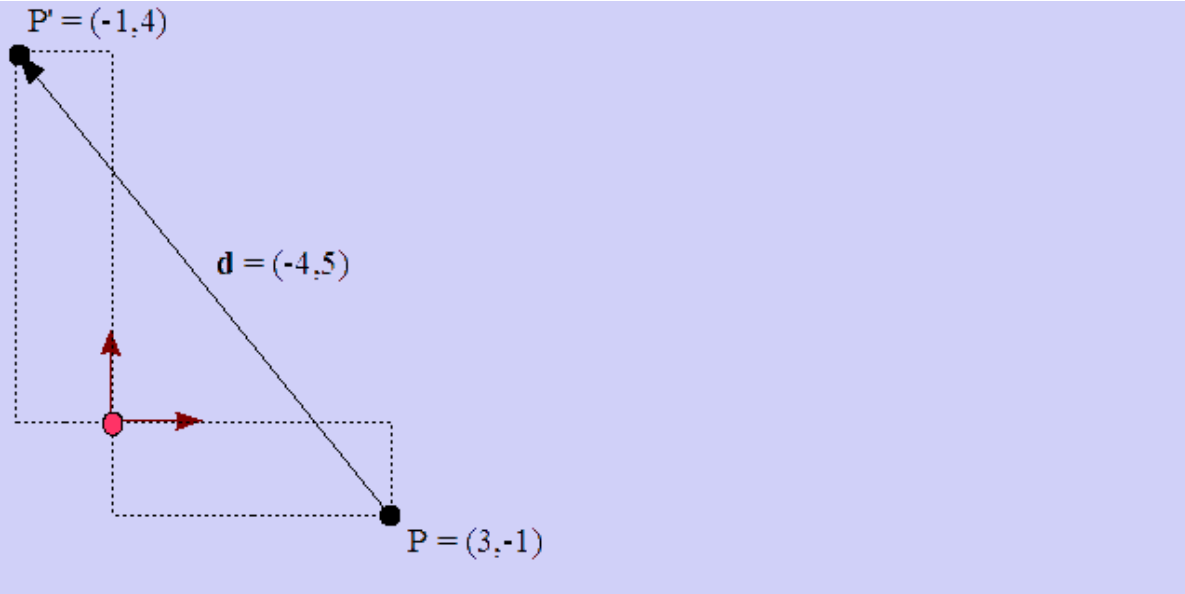


Figura 12 - Traslación de P=(3,-1) por el vector d=(-4,5) dando P′= (-1,4)

$$\mathbf{T} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ d_x & d_y & 1 \end{pmatrix}$$

$$\mathbf{T}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -d_x & -d_y & 1 \end{pmatrix}$$

Veamos un ejemplo sencillo, desplazando un punto, P = (3, -1), una distancia de **d** = (-4, 5),

$$P' = P\ \mathbf{T}(-4,5)$$

$$\begin{pmatrix} P'_x & P'_y & 1 \end{pmatrix} = \begin{pmatrix} 3 & -1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -4 & 5 & 1 \end{pmatrix}$$

$$P' = \begin{pmatrix} -1 & 4 & 1 \end{pmatrix}$$

Cambio de Escala

Esta operación sirve para modificar proporcional, pero no necesariamente uniformemente, los valores que representan los puntos o vectores a través de dos factores; uno para cada dimensión. Este factor multiplica cada valor de la entidad - punto o vector - en cuestión aumentando o reduciendo tal valor. Al aplicar esta operación a un objeto compuesto por segmentos, el resultado visual será el de un cambio de tamaño. Algebraicamente, expresamos esta operación de la siguiente manera:

$$P'_x = P_x * s_x$$
$$P'_y = P_y * s_y$$

donde, P contiene las coordenadas del punto original, P′ las del punto transformado, y s_x y s_y son los factores para el cambio de escala:

$$P = (P_x \quad P_y \quad 1)$$
$$P' = (P'_x \quad P'_y \quad 1)$$

En forma matricial, obtenemos la siguiente fórmula:

$$P' = P\,\mathbf{S}$$

donde,

$$\mathbf{S} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

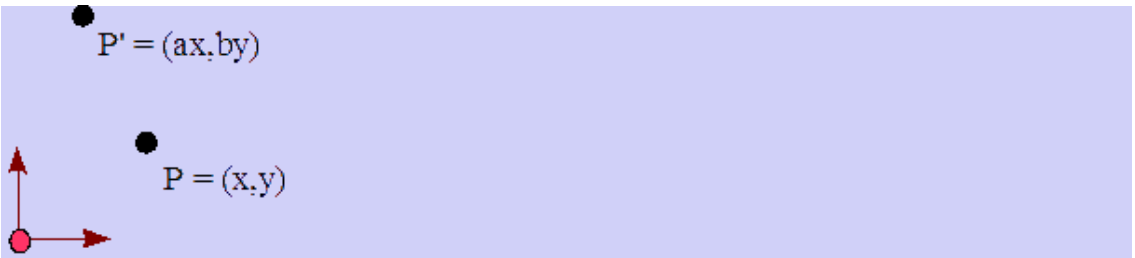


Figura 13 - Cambio de escala del punto P por los factores sx y sy

Como ocurre con la matriz de traslación, la matriz para el cambio de escala también tiene 2 grados de libertad. Asimismo podemos representar esta operación como una función que acepta dos parámetros: $\mathbf{S}(s_x,s_y)$.

Además de agrandar y reducir el tamaño, podemos crear el efecto de reflejo al usar números negativos. También hay que tener en cuenta que al multiplicar por un punto, éste puede cambiar de posición. Como esto no ocurre con el origen, (0,0), podemos multiplicar cualquier factor por este punto, sin variarlo. Por lo tanto, se suele elegir el origen para fijar un punto al aplicar un cambio de escala.

La inversa de una matriz de cambio de escala es simplemente la inversa de la multiplicación: la división. Esto es,

$$\mathbf{S} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{S}^{-1} = \begin{pmatrix} 1/s_x & 0 & 0 \\ 0 & 1/s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

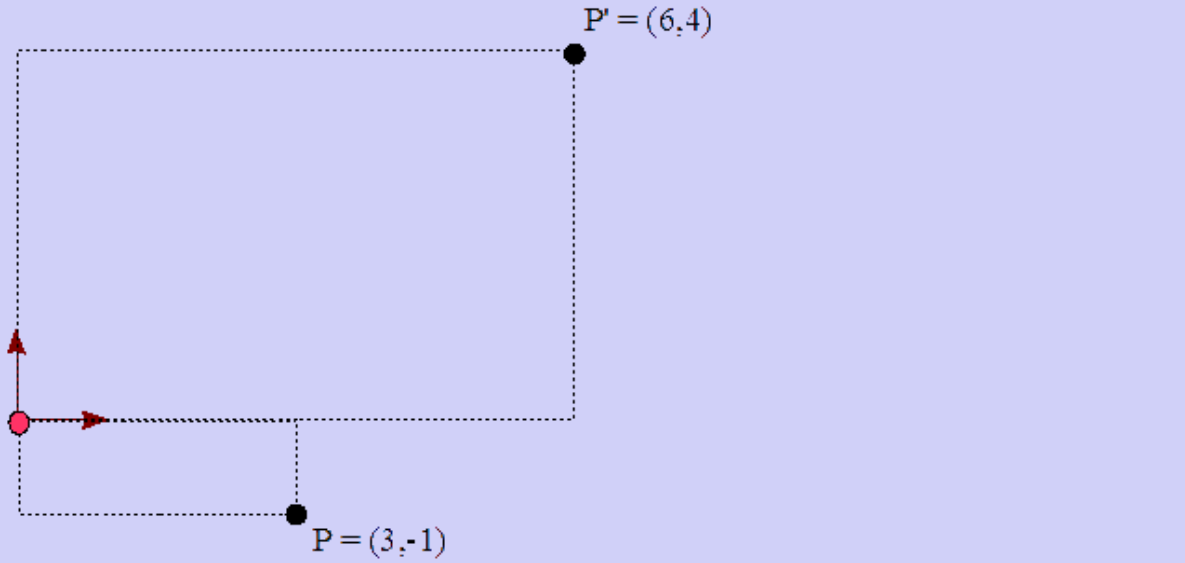


Figura 14 - Cambio de escala de P=(3,-1) por los factores sx=2 y sy=-3 dando P′=(6,4)

Ejemplo

Veamos un ejemplo, para aumentar y aplicar reflejo al siguiente punto, P = (3, -1), por los factores, $s_x = 2$ y $s_y = -4$,

$$P' = P\,\mathbf{S}(2,-4)$$

$$\begin{pmatrix} P'_x & P'_y & 1 \end{pmatrix} = \begin{pmatrix} 3 & -1 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 \\ 0 & -4 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$P' = \begin{pmatrix} 6 & 4 & 1 \end{pmatrix}$$

Rotación

En esta operación, el punto es transformado siguiendo el camino de una circunferencia con el origen como su centro. El camino a recorrer

depende del ángulo indicado. Las siguientes ecuaciones describen esta operación usando forma polar,

$$P_x = r * \cos \alpha$$
$$P_y = r * \operatorname{sen} \alpha$$

$$P'_x = r * \cos(\alpha + \beta)$$
$$P'_y = r * \operatorname{sen}(\alpha + \beta)$$

donde, P es el punto original, P′ el punto transformado, *r* es el radio en forma polar, y *α* y *β* son los ángulos inicial y final, respectivamente:

$$P = (P_x \;\; P_y \;\; 1)$$
$$P' = (P'_x \; P'_y \; 1)$$

Usando identidades trigonométricas, sustituimos los cosenos y senos de una suma de ángulos por expresiones equivalentes. Obtenemos la siguiente ecuación,

$$P'_x = r * \cos \alpha \cos \beta - r * \operatorname{sen} \alpha \operatorname{sen} \beta$$
$$P'_y = r * \operatorname{sen} \alpha \cos \beta + r * \cos \alpha \operatorname{sen} \beta$$

Aplicando las definiciones anteriores a estas ecuaciones, obtenemos que,

$$P'_x = P_x \cos \beta - P_y \operatorname{sen} \beta = P_x \cos \beta - P_y \operatorname{sen} \beta$$
$$P'_y = P_y \cos \beta + P_x \operatorname{sen} \beta = P_x \operatorname{sen} \beta + P_y \cos \beta$$

En forma matricial, obtenemos la siguiente fórmula:

$$P' = P \; \mathbf{R}$$

donde,

$$\mathbf{R} = \begin{pmatrix} \cos \beta & \operatorname{sen} \beta & 0 \\ -\operatorname{sen} \beta & \cos \beta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Esta matriz de rotación supone 4 grados de libertad y sólo requiere un parámetro: el ángulo de rotación. El ángulo se interpreta como positivo en el sentido contrario de las agujas del reloj a partir del eje X del marco de referencia. Podemos definir la rotación como una función, escribiendo: **R**(β).

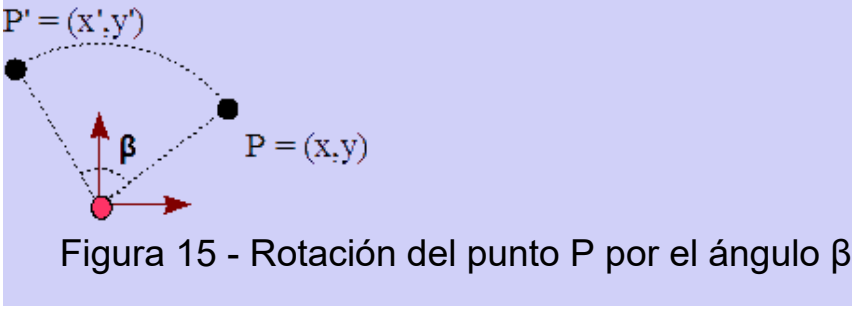


Figura 15 - Rotación del punto P por el ángulo β

La inversa de una rotación es sencillamente otra rotación con el mismo ángulo pero negativo, para indicar un sentido contrario del ángulo original. Esto es,

$$\mathbf{R} = \begin{pmatrix} \cos\beta & \operatorname{sen}\beta & 0 \\ -\operatorname{sen}\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}^{-1} = \begin{pmatrix} \cos -\beta & \operatorname{sen} -\beta & 0 \\ -\operatorname{sen} -\beta & \cos -\beta & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \cos\beta & -\operatorname{sen}\beta & 0 \\ \operatorname{sen}\beta & \cos\beta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Ejemplo

Vea mos un ejemplo para rotar el siguient e punto,

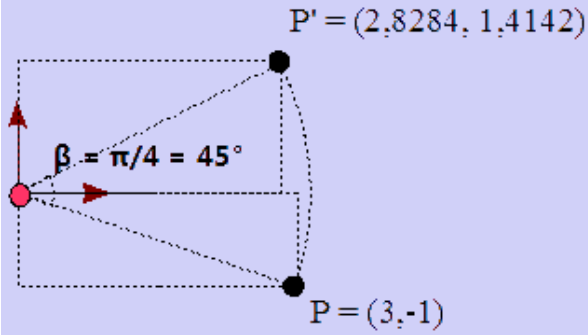


Figura 16 - Rotación del punto P=(3,-1) por el ángulo β=π/4

P = (3, -1), por el ángulo β = π/4 (45°).

$$P' = P\,\mathbf{R}(\pi/4)$$

$$
\begin{pmatrix} P'_x & P'_y & 1 \end{pmatrix} = \begin{pmatrix} 3 & -1 & 1 \end{pmatrix} \begin{pmatrix} \cos \pi/4 & \operatorname{sen} \pi/4 & 0 \\ -\operatorname{sen} \pi/4 & \cos \pi/4 & 0 \\ 0 & 0 & 1 \end{pmatrix}
$$

$$
P' = \begin{pmatrix} 3 & -1 & 1 \end{pmatrix} \begin{pmatrix} \sqrt{2}/2 & \sqrt{2}/2 & 0 \\ -\sqrt{2}/2 & \sqrt{2}/2 & 0 \\ 0 & 0 & 1 \end{pmatrix}
$$

$$
P' = \begin{pmatrix} 2{,}8284 & 1{,}4142 & 1 \end{pmatrix}
$$

Sesgado

La transvección o el sesgado trata de mudar todas las entidades en la misma dirección a lo largo de un eje. Esta combinación de traslación y cambio de escala "deforma" las entidades, tirando de ellas en ambos sentidos en la misma dirección. La deformación se basa en el ángulo, *α*, formado por el eje elegido y la línea que corta el eje elegido y contiene los nuevos puntos transformados. Las ecuaciones que describen esta operación a lo largo del eje X son las siguientes,

$$
\begin{aligned} P'_x &= P_x + P_y \operatorname{cotg} \alpha \\ P'_y &= P_y \end{aligned}
$$

donde,

$$
\begin{aligned} P &= \begin{pmatrix} P_x & P_y & 1 \end{pmatrix} \\ P' &= \begin{pmatrix} P'_x & P'_y & 1 \end{pmatrix} \end{aligned}
$$

Esto nos lleva a la siguiente forma matricial,

$$
P' = P \, \mathbf{H_x}
$$

donde,

$$
\mathbf{H}_x = \begin{pmatrix} 1 & 0 & 0 \\ \operatorname{cotg} \alpha & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}
$$

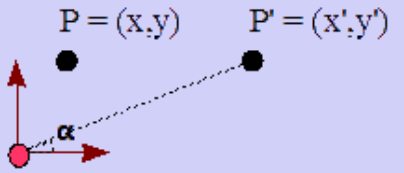


Figura 17 - Sesgado del punto P por el eje X con ángulo α

Podemos observar perfectamente que esta matriz sólo tiene 1 grado de libertad. Debemos tener cuidado con los valores de *α*, ya que la cotangente puede dispararse hacia el infinito, si el ángulo es 0 ó π (180°). La restricción para el ángulo es la siguiente: $0 < \alpha < \pi$ y $-\pi < \alpha < 0$. No tiene sentido que el ángulo se salga de este intervalo, pero sí hay que tenerlo presente. También observamos que no existe un sesgado con un ángulo de 90°. Esta transformación se puede describir como una función: $\mathbf{H_x}(\alpha)$.

Realizar la operación inversa de un sesgado supone otro sesgado con el mismo ángulo en forma negativa. Esto es,

$$\mathbf{H}_x = \begin{pmatrix} 1 & 0 & 0 \\ \cot g\ \alpha & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{H}_x^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ \cot g\ -\alpha & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ -\cot g\ \alpha & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Para conseguir la matriz para el sesgado a lo largo del eje Y, derivamos las ecuaciones de una manera muy parecida:

$$P'_x = P_x$$
$$P'_y = P_x \cot g\ \alpha + P_y$$

Obtendríamos la siguiente matriz,

$$\mathbf{H}_y = \begin{pmatrix} 1 & \cot g\ \alpha & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Ejemplo

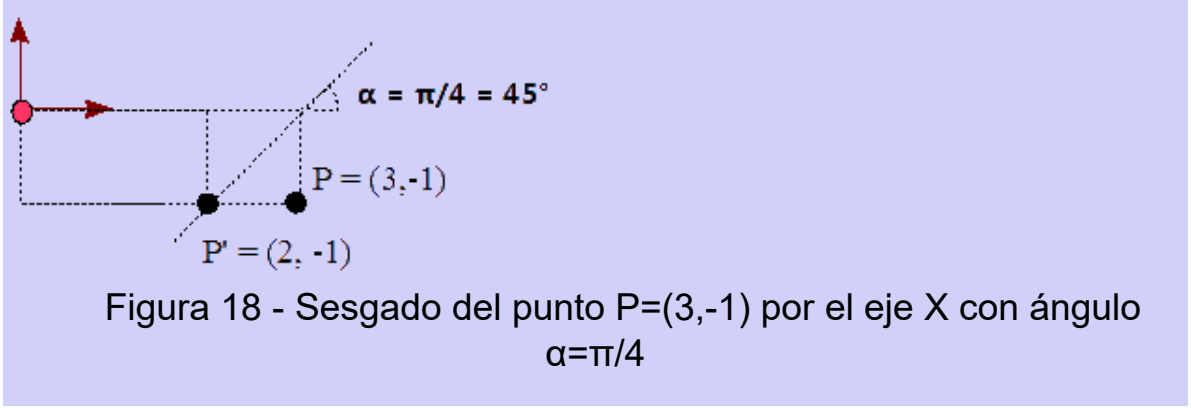


Figura 18 - Sesgado del punto P=(3,-1) por el eje X con ángulo α=π/4

Veamos un ejemplo sesgando el punto, P = (3, -1), por el eje X formando el ángulo α = π/4 (45°).

$$P' = P\ \mathbf{H}_x(\pi/4)$$

$$(\ P'_x\ P'_y\ 1\ ) = (\ 3\ -1\ 1\ )\ \begin{pmatrix} 1 & 0 & 0 \\ \cot g\ \pi/4 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$P' = (\ 3\ -1\ 1\ )\ \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$P' = (\ 2\ -1\ 1\ )$$

Uso de Transformaciones

Las transformaciones básicas que se han visto hasta ahora controlan las entidades - puntos y vectores - que operan, con resultados diferentes. Analizando estas transformaciones, podemos clasificarlas según el efecto, que provocan, de las entidades resultantes. Se suele hablar de transformaciones de "cuerpo rígido", que son aquellas transforamciones cuyos cuerpos no sufren deformaciones. Cualquier secuencia de traslaciones y rotaciones cumple con este criterio, ya que no distorsiona el objeto, conservando el paralelismo y las longitudes de las líneas al igual que los ángulos. Sin embargo, operaciones que incluyan cambios de escala y sesgados moldean los cuerpos, por lo que no conservan las

longitudes de las líneas ni los ángulos. Aun existiendo estas diferencias, sí podemos garantizar que cualquier transformación afín conserva el paralelismo de las líneas.

Con estas transformaciones, podemos crear nuevas operaciones compuestas por las operaciones básicas. Para llevar a cabo esta composición, aplicamos una serie de transformaciones multiplicándolas por cada entidad a transformar. Multiplicamos la matriz, que representa la transformación, por el punto o vector. Esto es,

$$
\begin{aligned}
P' &\leftarrow P\ \mathbf{M}_0 \\
P' &\leftarrow P'\ \mathbf{M}_1 \\
P' &\leftarrow P'\ \mathbf{M}_2 \\
P' &\leftarrow P'\ \mathbf{M}_3 \\
P' &\leftarrow P'\ \mathbf{M}_4 \\
&\vdots \\
P' &\leftarrow P'\ \mathbf{M}_n
\end{aligned}
$$

Como se trata de multiplicaciones de varias transformaciones, $\mathbf{M}_n$, aplicadas a los mismos puntos original y transformados, P, podemos agrupar estas matrices de transformaciones en una serie de multiplicaciones. Esto es,

$$P \leftarrow P\ \mathbf{M}_0\ \mathbf{M}_1\ \mathbf{M}_2\ \ldots\ \mathbf{M}_n$$

Podemos agregar más matrices al producto para aplicar más transformaciones, creando una encadenación. Al multiplicar todas las matrices de transformación, terminaremos con una sola matriz que recoge todas las transformaciones aplicadas a la entidad en cuestión. De esta manera, podemos multiplicar varias matrices iguales para luego multiplicar la transformación resultante a todas las entidades que nos interesan. Esto optimiza la cantidad de multiplicaciones que debemos hacer. Por ejemplo,

$$\mathbf{M} \leftarrow \mathbf{M}_0\ \mathbf{M}_1\ \mathbf{M}_2\ \ldots\ \mathbf{M}_n$$

$$
\begin{aligned}
P'_0 &\leftarrow P_0\ \mathbf{M} \\
P'_1 &\leftarrow P_1\ \mathbf{M}
\end{aligned}
$$

$$
\begin{aligned}
P'_2 &\leftarrow P_2\ \mathbf{M} \\
P'_3 &\leftarrow P_3\ \mathbf{M} \\
&\ \ \vdots \\
P'_n &\leftarrow P_n\ \mathbf{M}
\end{aligned}
$$

Composición

Veamos algunas transformaciones compuestas que nos pueden ser útiles en el manejo de nuestros modelos 2D. Debemos tener presente que una secuencia de transformaciones no es necesariamente igual a la misma secuencia en diferente orden. Por ejemplo, la secuencia **T R T S** ≠ **S T T R** ≠ **S T R T**. Obviamente, esto tiene sentido, ya que no existe la propiedad conmutativa para las matrices.

El primer ejemplo que trataremos es rotar un objeto por un punto fijo. Como ya hemos explicado, la rotación se realiza por el origen (0,0). Para rotar por otro punto, simplemente mudamos ese punto fijo al origen, rotamos por el origen, y luego devolvemos el punto a las coordenadas del punto fijo. Algebraicamente, representamos esta idea de la siguiente manera:

$$\mathbf{M} = \mathbf{T}(-p_f)\ \mathbf{R}(\alpha)\ \mathbf{T}(p_f)$$

donde, $p_f = (x_f, y_f)$ es el punto fijo por el que queremos rotar y α es el ángulo de rotación.

Podemos multiplicar todas las matrices para obtener la matriz resultante, **M**, que es la siguiente:

$$\mathbf{M} = \begin{pmatrix} \cos\alpha & \operatorname{sen}\alpha & 0 \\ -\operatorname{sen}\alpha & \cos\alpha & 0 \\ (x_f - x_f\cos\alpha + y_f \operatorname{sen}\alpha) & (y_f - x_f \operatorname{sen}\alpha - y_f\cos\alpha) & 1 \end{pmatrix}$$

Ejemplos

Veamos unos ejemplos prácticos.

1. Tenemos dos puntos, p = (4,2) y q = (-3,1). Queremos rotar estos dos puntos alrededor del punto (fijo), F = (2,-1), por un ángulo, $\alpha = \pi/6$ (30°),

$$p' = p\ \mathbf{T}(\ (-2,1)\ )\ \mathbf{R}(\pi/6)\ \mathbf{T}(\ (2,-1)\ )$$
$$q' = q\ \mathbf{T}(\ (-2,1)\ )\ \mathbf{R}(\pi/6)\ \mathbf{T}(\ (2,-1)\ )$$

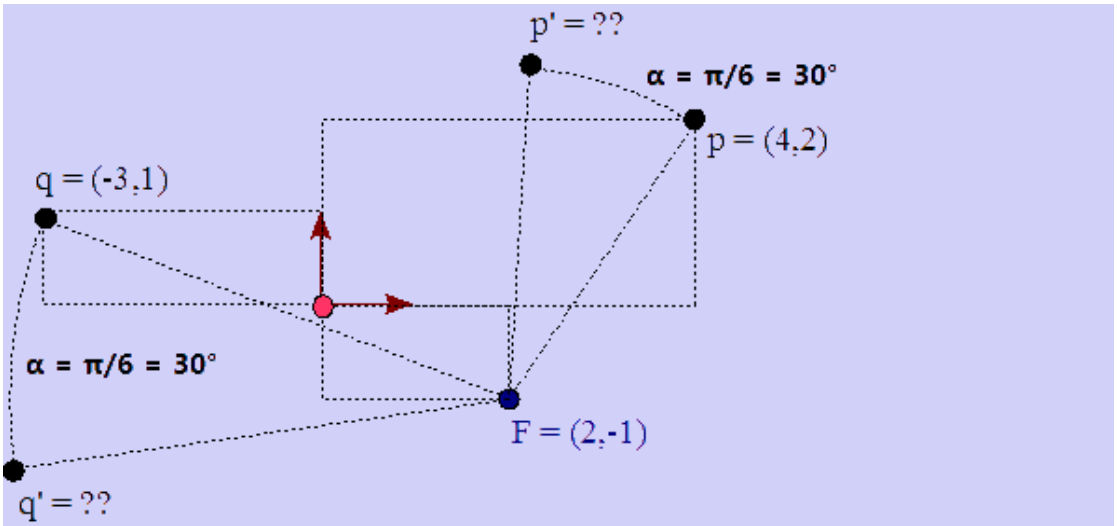


Figura 19 - Rotación de los puntos p y q por el punto F con ángulo α

Calculemos la matriz resultante, **M**,

$$\mathbf{M} = \begin{pmatrix} \cos \pi/6 & \operatorname{sen} \pi/6 & 0 \\ -\operatorname{sen} \pi/6 & \cos \pi/6 & 0 \\ (2 - 2\cos \pi/6 - \operatorname{sen} \pi/6) & (-1 - 2\operatorname{sen} \pi/6 + \cos \pi/6) & 1 \end{pmatrix}$$

Aplicando la matriz resultante, obtenemos el siguiente paso,

$$p' = \begin{pmatrix} 4 & 2 & 1 \end{pmatrix} \begin{pmatrix} 0{,}8660 & 0{,}5 & 0 \\ -0{,}5 & 0{,}8660 & 0 \\ -0{,}2321 & -1{,}1340 & 1 \end{pmatrix}$$

$$q' = \begin{pmatrix} -3 & 1 & 1 \end{pmatrix} \begin{pmatrix} 0{,}8660 & 0{,}5 & 0 \\ -0{,}5 & 0{,}8660 & 0 \\ -0{,}2321 & -1{,}1340 & 1 \end{pmatrix}$$

Los resultados son los siguientes,

```
p' = (  2,2321,  2,5981 )
q' = ( -3,3301, -1,7680 )
```

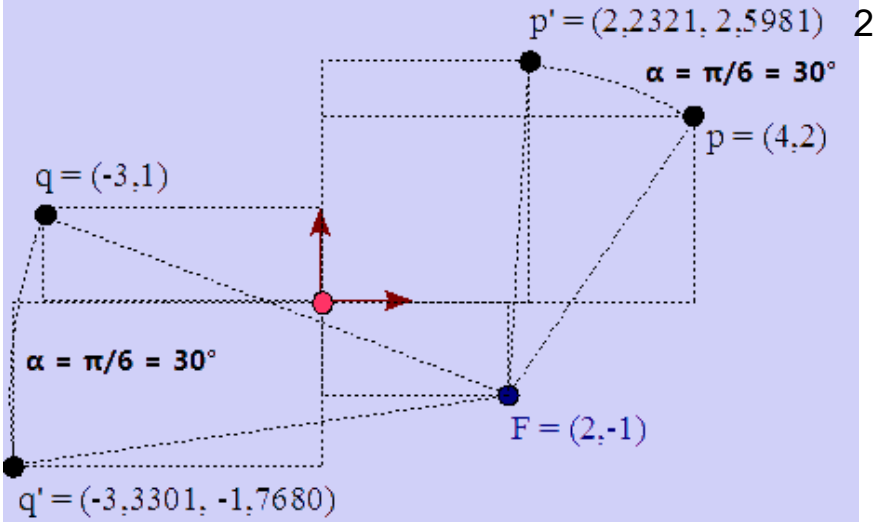


Figura 20 - Los puntos p′ y q′

2. Otro ejemplo parecido al anterior es a la hora de agrandar o empequeñecer un objeto el cual está compuesto por varios puntos y líneas. Esto supone un cambiar de escala, pero como hemos visto anteriormente, esta operación también realiza una traslación proporcional a los parámetros del cambio de escala. La solución es mudar el objeto al origen, realizar la operación del cambio de escala, para luego regresar el mismo objeto a su posición original. Esto supone que debemos elegir un punto fijo e invariable para poder trasladar y fijar el objeto al origen. Veamos las operaciones matriciales básicas para implementar esta operación compleja:

$$\mathbf{M} = \mathbf{T}(-p_f)\ \mathbf{S}(s_x, s_y)\ \mathbf{T}(p_f)$$

donde, $p_f = (x_f, y_f)$ es el punto fijo por el que queremos cambiar el tamaño de cualquier entidad según los factores indicados por s_x y s_y.

Multiplicando todas estas matrices obtenemos la siguiente matriz resultante, **M**,

$$\begin{pmatrix} s_x & 0 & 0 \end{pmatrix}$$

$$\mathbf{M} = \begin{pmatrix} 0 & s_y & 0 \\ x_f(1 - s_x) & y_f(1 - s_y) & 1 \end{pmatrix}$$

3. Veamos un ejemplo de esta transformación.

Tenemos un cuadrilátero formado por cuatro puntos, p = (4,2), q = (-3,1), r = (-2,-2), y s = (3,0). Vamos a agrandar este objeto desproporcionalmente: $s_x = 2$ y $s_x = 0{,}5$ y fijado en el punto, F = (1,2). Algebraicamente, tenemos lo siguiente,

$$\begin{aligned} p' &= p\,\mathbf{T}(-1,-2)\,\mathbf{S}(2,\ 0{,}5)\,\mathbf{T}(1,2) \\ q' &= q\,\mathbf{T}(-1,-2)\,\mathbf{S}(2,\ 0{,}5)\,\mathbf{T}(1,2) \\ r' &= r\,\mathbf{T}(-1,-2)\,\mathbf{S}(2,\ 0{,}5)\,\mathbf{T}(1,2) \\ s' &= s\,\mathbf{T}(-1,-2)\,\mathbf{S}(2,\ 0{,}5)\,\mathbf{T}(1,2) \end{aligned}$$

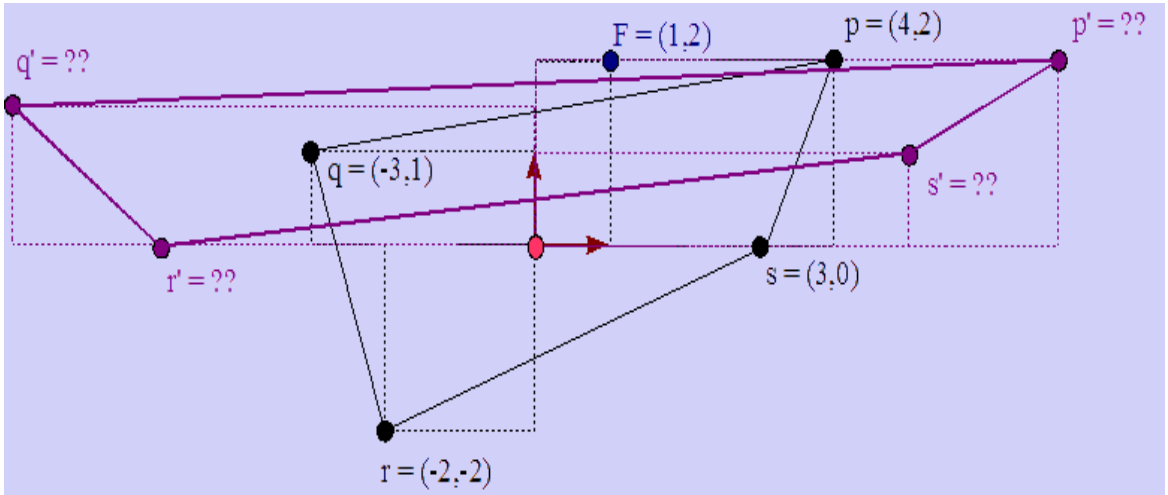


Figura 21 - Agrandamos desproporcionalmente un cuadrilátero fijado en F=(1,2)

Calculemos la matriz resultante, **M**,

$$\mathbf{M} = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 0{,}5 & 0 \\ -1 & 1 & 1 \end{pmatrix}$$

Ahora multiplicamos cada punto, que describe nuestro objeto, por la matriz resultante,

$$p' = \begin{pmatrix} 4 & 2 & 1 \end{pmatrix} \begin{pmatrix} 2 & 0 & 0 \\ 0 & 0{,}5 & 0 \end{pmatrix}$$

```
                    ( -1   1     1 )

                    (  2   0     0 )
q' = ( -3   1   1 ) (  0   0,5   0 )
                    ( -1   1     1 )

                    (  2   0     0 )
r' = ( -2  -2   1 ) (  0   0,5   0 )
                    ( -1   1     1 )

                    (  2   0     0 )
s' = (  3   0   1 ) (  0   0,5   0 )
                    ( -1   1     1 )
```

Al final, obtenemos la siguiente información,

```
p' = (  7, 2 )
q' = ( -7, 1,5 )
r' = ( -5, 0 )
s' = (  5, 1 )
```

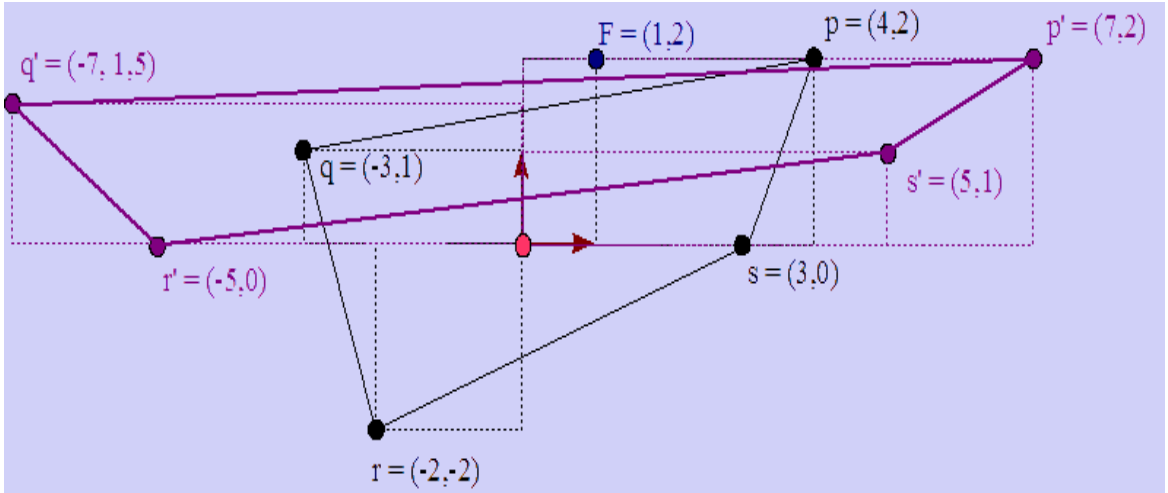


Figura 22 - El nuevo cuadrilátero formado por los puntos p′, q′, r′, y s′

I nstancia

Otro claro ejemplo del uso de las transformaciones tiene que ver con la creación de nuevos objetos. En muchas ocasiones, queremos mostrar varios objetos basados en una misma figura. Sería algo engorroso describir las coordenadas exactas que describen cada instancia basada en la misma figura. Por ello, definimos un juego de diferentes objetos

para luego crear cada objeto escogido de nuestro juego, que queremos mostrar en nuestra escena. Usando traslaciones, podemos colocar todas las instancias de cualesquier objetos en nuestra escena los cuales pueden estar descritos de una sola manera. Por supuesto, también podemos modificar cada instancia antes de colocarla en la escena. Esto supone aplicar otras transformaciones a nuestros objetos antes de trasladarlos a sus respectivas posiciones finales.

Ejemplo

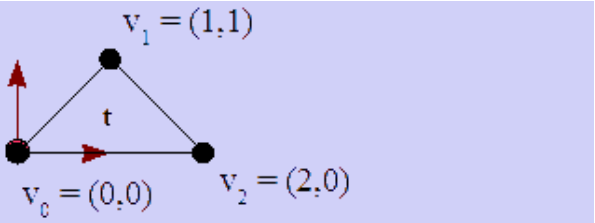


Figura 23 - El triángulo original, t

Describimos un triángulo, *t*, con la siguiente lista de vértices: v = ((0,0), (1,1), y (2,0)). Vamos a crear cuatro triángulos basados en esta descripción: t_0, t_1, t_2, y t_3. Sin embargo, aplicaremos varias transformaciones a cada uno de los triángulos para colocarlos en nuestra escena. Describamos nuestras intenciones algebraicamente,

t_0 = t **S**(-1,2) **R**(π/6) **T**((3,4))
t_1 = t **R**(-π/6) $\mathbf{H_x}$(π/12) **T**((1,-6))
t_2 = t $\mathbf{H_x}$(π/6) $\mathbf{H_y}$(π/3) **S**(0,25, 2) **T**((-5,0))
t_3 = t **S**(-1,-0,2) **T**((-3,-2))

Las matrices resultantes para cada serie de transformaciones son las siguientes:

$$M_0 = \begin{pmatrix} -0,8660 & -0,5 & 0 \\ -1 & 1,7321 & 0 \\ 3 & 4 & 1 \end{pmatrix}$$

$$M_1 = \begin{pmatrix} -1 & -0,5 & 0 \\ 3,7321 & 0,8660 & 0 \\ 1 & -6 & 1 \end{pmatrix}$$

$$M_2 = \begin{pmatrix} 0,25 & 1,1547 & 0 \\ 0,4330 & 4 & 0 \end{pmatrix}$$

$$M_3 = \begin{pmatrix} -5 & 0 & 1 \\ -0{,}125 & 0 & 0 \\ 0 & -0{,}2 & 0 \\ -6 & -5 & 1 \end{pmatrix}$$

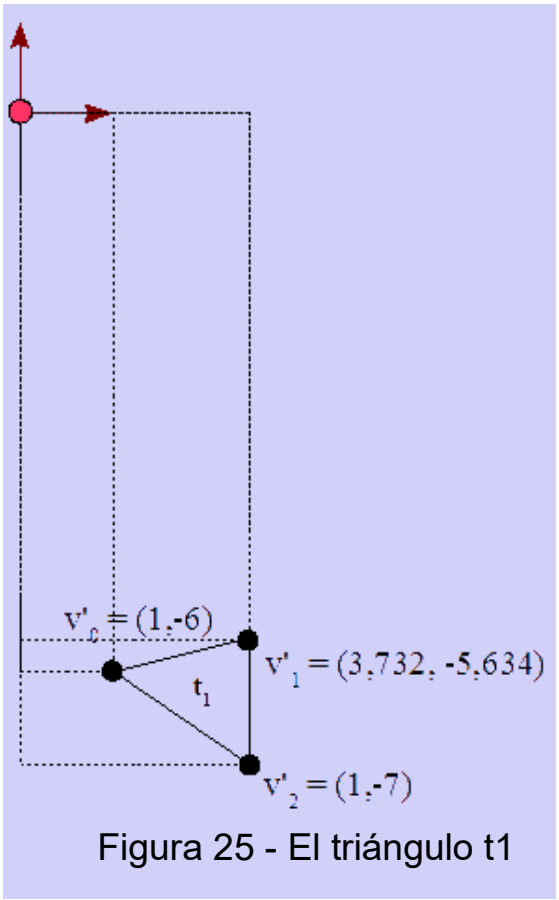


Figura 25 - El triángulo t1

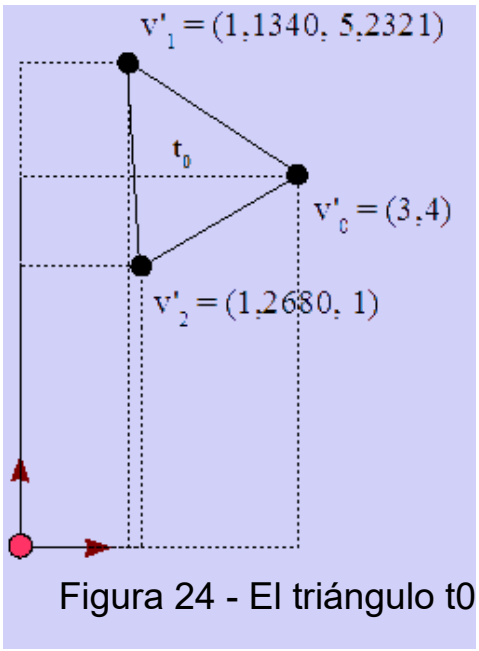


Figura 24 - El triángulo t0

Acabaremos con los siguientes cálculos:

$t_0{:}v_0 = (\ 0\ \ 0\ \ 1\ )\ M_0$
$t_0{:}v_1 = (\ 1\ \ 1\ \ 1\ )\ M_0$
$t_0{:}v_2 = (\ 2\ \ 0\ \ 1\ )\ M_0$

$t_1{:}v_0 = (\ 0\ \ 0\ \ 1\ )\ M_1$
$t_1{:}v_1 = (\ 1\ \ 1\ \ 1\ )\ M_1$
$t_1{:}v_2 = (\ 2\ \ 0\ \ 1\ )\ M_1$

$t_2{:}v_0 = (\ 0\ \ 0\ \ 1\ )\ M_2$
$t_2{:}v_1 = (\ 1\ \ 1\ \ 1\ )\ M_2$
$t_2{:}v_2 = (\ 2\ \ 0\ \ 1\ )\ M_2$

$t_3{:}v_0 = (\ 0\ \ 0\ \ 1\ )\ M_3$
$t_3{:}v_1 = (\ 1\ \ 1\ \ 1\ )\ M_3$
$t_3{:}v_2 = (\ 2\ \ 0\ \ 1\ )\ M_3$

Como se puede ver, debemos aplicar las transformaciones a todos los puntos que representan los vértices de cada triángulo. Al final, obtendremos la siguiente información para los cuatro triángulos:

$t_0{:}v_0 = (\ 3\ \ 4\ \ 1\ )$
$t_0{:}v_1 = (\ 1{,}1340\ \ 5{,}2321\ \ 1\ )$
$t_0{:}v_2 = (\ 1{,}2680\ \ 3\ \ 1\ )$

$t_1{:}v_0 = (\ 1\ -6\ \ 1\ )$
$t_1{:}v_1 = (\ 3{,}7321\ -5{,}634\ \ 1\ )$
$t_1{:}v_2 = (\ 1\ -7\ \ 1\ )$

```
t2:v0 = ( -5   0   1 )
t2:v1 = ( -4,3170   5,1547   1 )
t2:v2 = ( -4,5   2,3094   1 )

t3:v0 = ( -3  -2   1 )
t3:v1 = ( -4  -2,2   1 )
t3:v2 = ( -5  -2   1 )
```

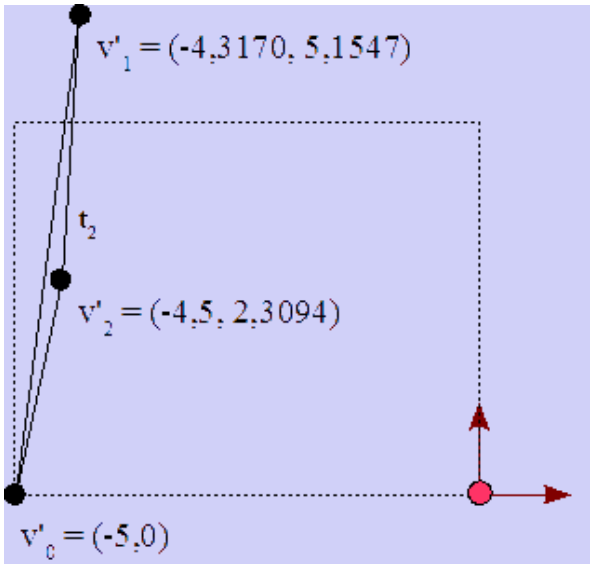


Figura 26 - El triángulo t2

Modelado

Hasta ahora hemos visto la aplicación de las transformaciones afines a puntos. Sin embargo, nuestras escenas seguramente no se compondrán de puntos individuales, sino de objetos compuestos de segmentos. Asimismo, estos segmentos se compondrán de puntos. Como no podemos manipular todos los puntos individualmente en la práctica, sí podemos manipular los puntos finales del segmento. Por lo tanto, describiremos nuestros objetos según los puntos y los segmentos conectados entre ellos.

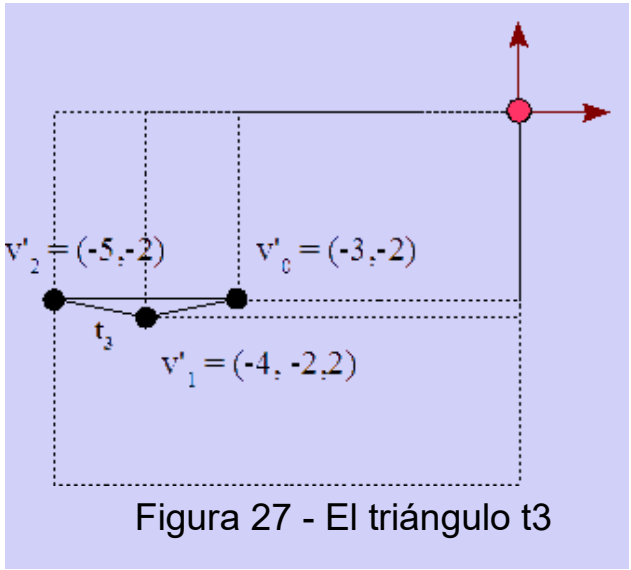
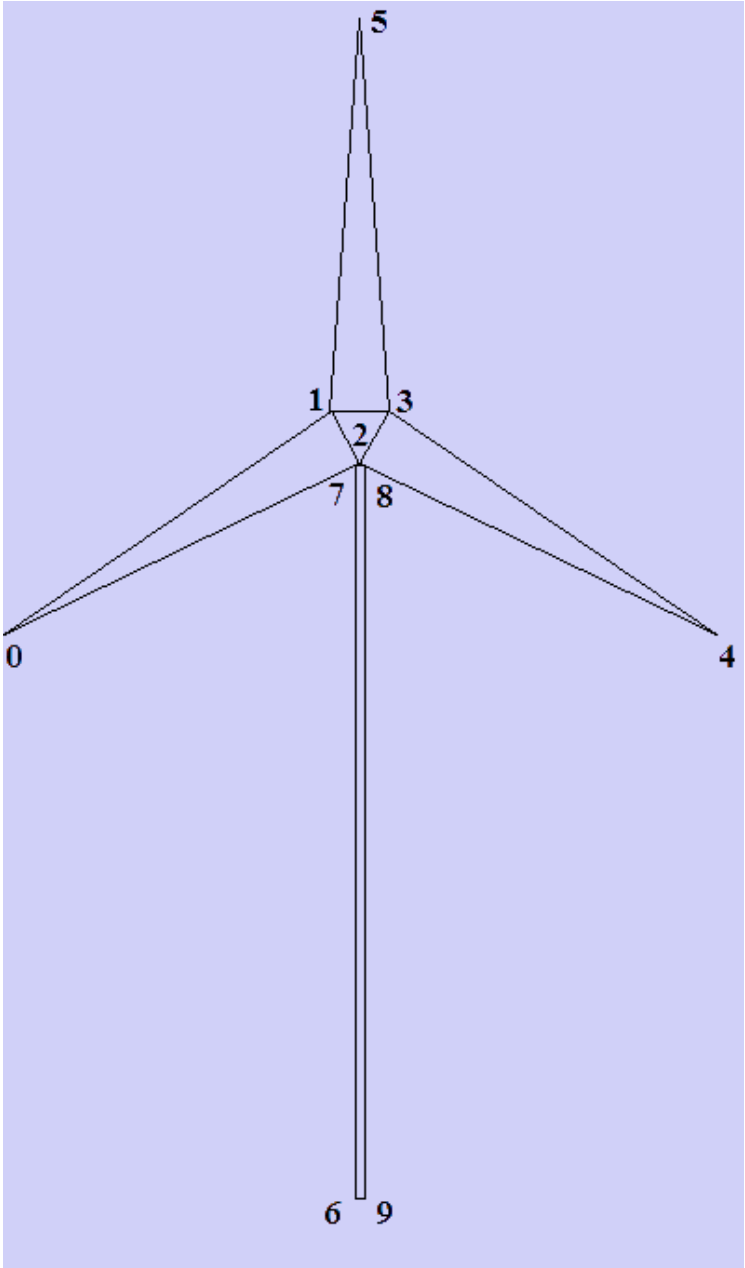


Figura 27 - El triángulo t3

Representación

Para describir cada objeto, necesitaremos una forma de organizar la información: los puntos y segmentos. Además, capturaremos la geometría - puntos y segmentos - y la topología: la asociación de los puntos para formar la figura. Si pensamos jerárquicamente, daremos con la idea de descomponer un objeto geométrico en otros objetos más pequeños y fundamentales. Estos objetos fundamentales pueden ser

representados como polígonos. Cada polígono a su vez se forma de vértices y aristas. Para describir las aristas de un polígono, crearemos una lista de vértices. De esta manera, optimizamos la cantidad de vértices que usaremos. Como seguramente tendremos polígonos que coincidan en una o varias aristas de otros polígonos, entonces también existirán vértices que coinciden. Por ello, es óptimo guardar una copia de cada vértice y luego referirnos a cada uno, en lugar de copiarlos. Podemos ver esta estrategia en el siguiente esquema acerca de la organización jerárquica de la información que describe un molino.



Un matiz importante a notar es el orden de los vértices para representar cada polígono. Hemos escogido los vértices contiguos en el sentido de las agujas del reloj. Esta elección se suele llamar operaciones de mano izquierda, porque podemos usar los cuatro dedos de la mano izquierda para seguir las aristas de un polígono. Este método tiene que ver con la representación que hemos hecho hasta estos momentos al igual que el álgebra matricial usada. Esto también tiene importancia cuando pasemos al modelado en 3D.

Orden de la Representación

Figura 28 - Molino

Tenemos dos formas de representar la información en las matrices y vectores. Los elementos de cada fila pueden representar los valores para las coordenadas de los ejes X e Y. Por otro lado, también podríamos guardar las coordenadas en las columnas. Esto supondría cambiar el orden de las multiplicaciones matriciales. Típicamente, hablamos de operaciones matriciales de fila, con los elementos en cada fila, y las de columna, donde la información se guarda en las columnas.

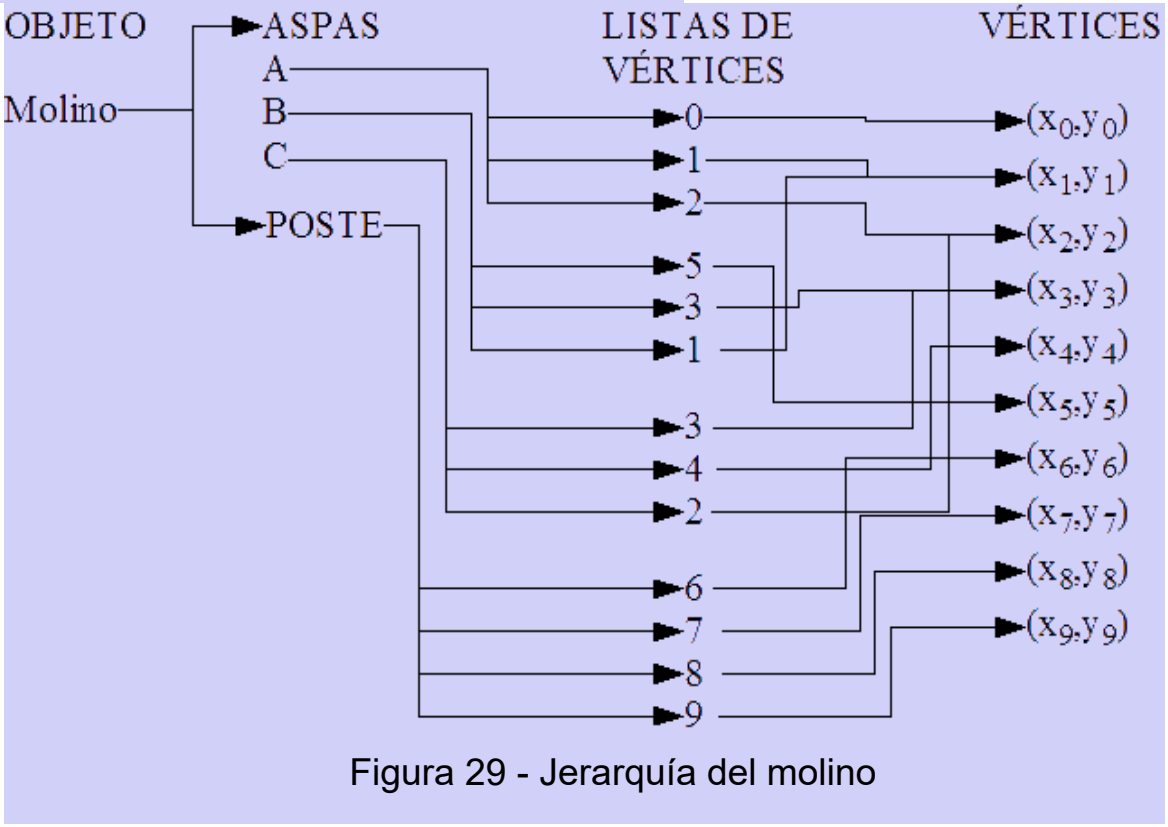


Figura 29 - Jerarquía del molino

Dependiendo del tipo de las operaciones matriciales, también nos interesa hablar de la manera de organizar la información de nuestros modelos. Geométricamente, tenemos que elegir los ejes y la relación entre ellos. Esta elección dictará la colocación y orientación de nuestros objetos geométricos. También definirá el sentido de las rotaciones. Por ello, solemos hablar de la regla de la mano derecha o de la mano izquierda. Esto hace alusión al uso de una mano u otra para determinar la relación entre los ejes y principalmente para las rotaciones. Como en dos dimensiones sólo tenemos dos ejes, no hablaremos de esta regla ya que

sólo existe una orientación positiva sobre los ejes. Volveremos a tratar este concepto cuando veamos el modelado en tres dimensiones.

Hasta ahora y en el resto de este curso, usaremos las operaciones matriciales de fila y la regla de la mano izquierda.

Vista

Después de todas las transformaciones, que nos interesan aplicar a los objetos en nuestra escena, tenemos que convertir nuestra escena al medio que queremos. Nuestra escena se compone de los objetos geométricos. Sin embargo, la parte que queremos ver - la vista - se basa en un área que envuelve estos objetos geométricos de la escena. Típicamente, querremos mostrar la vista, como un área rectangular, en toda la pantalla, en píxeles. Por supuesto, existirán ocasiones en las que elegiremos una parte de la pantalla que contendrá nuestra vista. El paso de nuestro mundo o universo en 2D a la pantalla en 2D no es uno complicado, pero sí es necesario.

Podemos afrontar este problema como un cambio de representación. Tenemos el marco de referencia de nuestro universo descrito como $\{\mathbf{v_x},\mathbf{v_y}, (0,0)\}$ y queremos representar los objetos geométricos en nuestra escena en el marco de referencia de la pantalla descrito como $\{\mathbf{u_x},\mathbf{u_y}, Q_0\}$. Las siguientes ecuaciones describen el marco de referencia de la pantalla en términos del marco de nuestro universo,

$$\mathbf{u_x} = s_x \mathbf{v_x}$$
$$\mathbf{u_y} = -s_y \mathbf{v_y}$$
$$Q_0 = q_x \mathbf{v_x} + q_y \mathbf{v_y}$$

donde,

s_x es el factor para el cambio de escala en el eje X; es decir, la proporción de anchuras de la escena y de la pantalla: d_x/d'_x

s_y es el factor para el cambio de escala en el eje Y; es decir, la proporción de anchuras de la escena y de la pantalla: d_y/d'_y

(q_x,q_y) es el punto en la esquina superior izquierda que encuaderna nuestra escena. Este punto representará el origen en la pantalla.

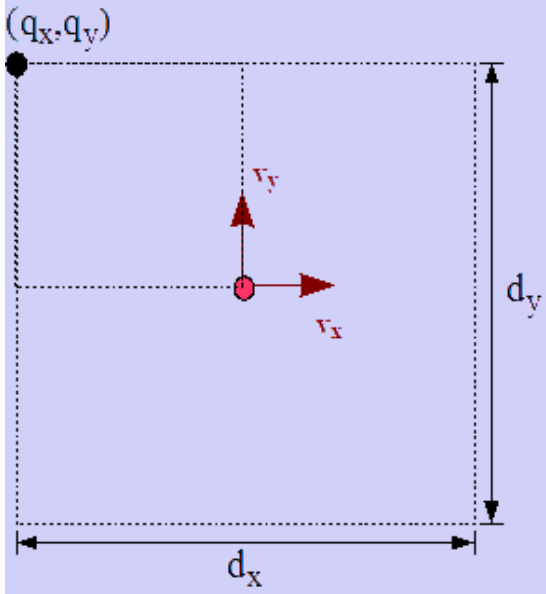


Figura 30 - Marco de referencia de nuestra escena

La matriz de representación es la siguiente,

$$\mathbf{M} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & -s_y & 0 \\ q_x & q_y & 1 \end{pmatrix}$$

Como queremos cambiar la representación, necesitaremos determinar la inversa de **M**, que es,

$$\mathbf{M}^{-1} = \begin{pmatrix} 1/s_x & 0 & 0 \\ 0 & -1/s_y & 0 \\ -q_x/s_x & q_y/s_y & 1 \end{pmatrix}$$

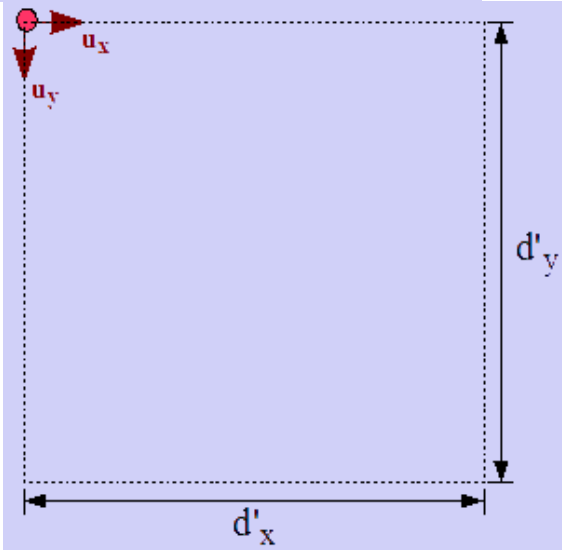


Figura 31 - Marco de referencia de la pantalla

Ejemplo

Tenemos dos triángulos, *s* y *t*, en nuestra escena. El marco de referencia que representa nuestro universo es el siguiente: {**i**,**j**, (0,0)}. La vista de nuestra escena será representada por el cuadrado cuya esquina superior izquierda es (-3,2) y cuyo lado mide 8. La siguiente figura 32 muestra nuestra escena y las dimensiones de nuestra vista en el marco de referencia del universo.

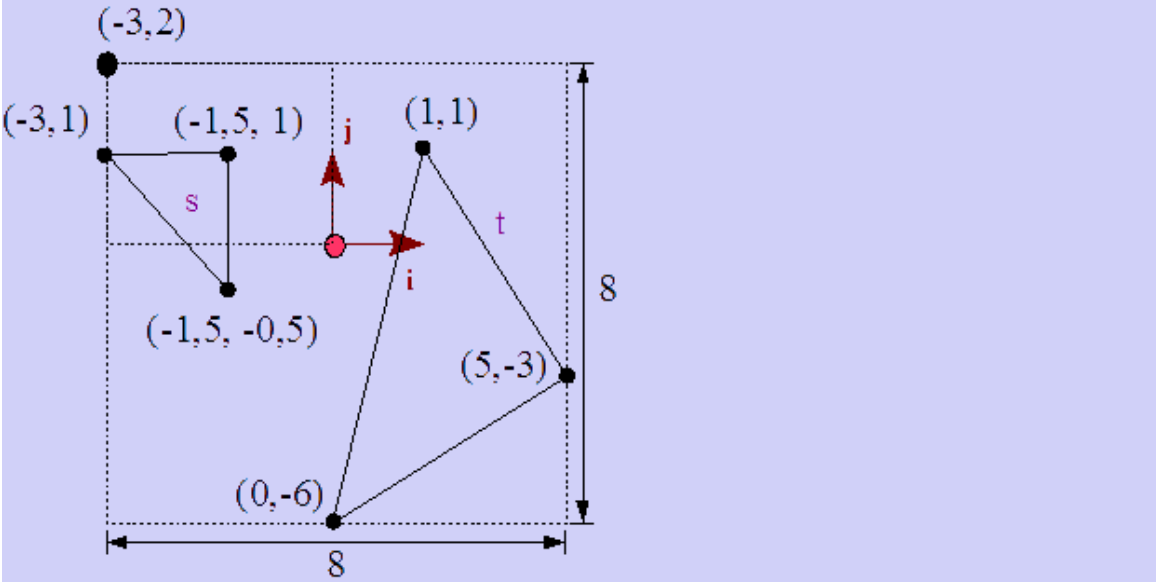


Figura 32 - Escena con dos triángulos y la vista de nuestra escena

Queremos representar estos dos triángulos en la pantalla cuya resolución es 400 x 400. Las siguientes ecuaciones describen el marco de referencia de la pantalla en términos del marco de nuestro universo,

$$400\ \mathbf{u_x} = 8\ \mathbf{i}$$
$$400\ \mathbf{u_y} = -8\ \mathbf{j}$$
$$Q_0 = -3\ \mathbf{i} + 2\ \mathbf{j}$$

Simplificando, terminamos con,

$$\mathbf{u_x} = 0{,}02\ \mathbf{i}$$
$$\mathbf{u_y} = -0{,}02\ \mathbf{j}$$
$$Q_0 = -3\ \mathbf{i} + 2\ \mathbf{j}$$

Obtenemos la siguiente matriz de representación,

$$\mathbf{M} = \begin{pmatrix} 0{,}02 & 0 & 0 \\ 0 & -0{,}02 & 0 \\ -3 & 2 & 1 \end{pmatrix}$$

Para poder cambiar la representación, necesitamos calcular la inversa de la matriz, **M**, que es,

$$\begin{pmatrix} 50 & 0 & 0 \end{pmatrix}$$

$$\mathbf{M}^{-1} = \begin{pmatrix} 0 & -50 & 0 \\ 150 & 100 & 1 \end{pmatrix}$$

Podemos ver el resultado del cambio de representación de los vértices de ambos triángulos en la figura 33.

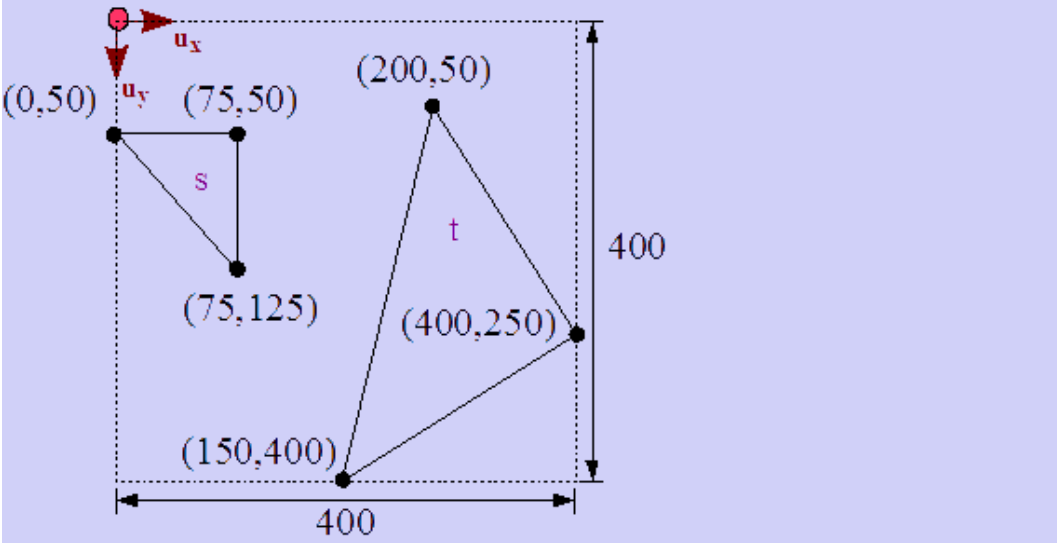


Figura 33 - Escena con dos triángulos en pantalla en píxeles

Otra forma de resolver el problema de conversión es aplicar una transformación compuesta de una traslación seguida de un cambio de escala. Trasladamos el punto en la esquina superior izquierda que de nuestra vista al origen. Luego, cambiamos de escala y con reflejo. Los factores de tal cambio de escala se basan en las proporciones de la anchura de la pantalla con respecto a la anchura de la vista para el eje X, y las proporciones de la altura de la pantalla con respecto a la altura de la vista para el eje Y. En forma matricial, tenemos la siguiente transformación afín,

$$\mathbf{M} = \mathbf{T}(-q_x,-q_y)\ \mathbf{S}(s_x,-s_y)$$

donde,
s_x es la proporción de las anchuras de la pantalla y de la escena: d'_x/d_x
s_y es la proporción de las alturas de la pantalla y de la escena: d'_y/d_y
(q_x,q_y) es el punto que representará el origen en la pantalla de la esquina superior izquierda del rectángulo que encuaderna nuestra escena.

Usando el ejemplo anterior, obtendríamos la siguiente matriz,

```
M = T(3,-2) S(400/8,-400/8)
```

Al final, conseguimos la misma matriz que la solución obtenida previamente al realizar un cambio de marco de referencia,

$$\mathbf{M}^{-1} = \begin{pmatrix} 50 & 0 & 0 \\ 0 & -50 & 0 \\ 150 & 100 & 1 \end{pmatrix}$$

Observaciones

La primera pregunta, que muchos iniciados se hacen, es *"¿de verdad necesitamos hacer todo esto?"*. Aunque todos estos conceptos y la mecánica presentada parezcan más complicados de lo que deberían ser, nos servirán cuando pasemos a manipular y operar con objetos en tres dimensiones. Por ahora, todas estas operaciones nos servirán de base. Esto no significa que no deberíamos usar estas operaciones en dos dimensiones, especialmente cuando tenemos objetos complicados.

En la práctica, todas estas operaciones matriciales y vectoriales ya vienen implementadas en software como bibliotecas y API's gráficas y en hardware como instrucciones de la GPU (unidad de procesamiento gráfico) de las tarjetas aceleradoras gráficas. Tanto a nivel de hardware como de software, seguimos una arquitectura de procesamiento en serie y por etapas para generar una imagen a partir de su representación geométrica. Empezamos con la geometría de los objetos, que principalmente es el grupo de vértices. El procesamiento total de estos datos geométricos se puede repartir y distribuir por etapas o fases. En estos momentos, tenemos el siguiente esquema de distribución de tareas.

Figura 34 - Procesamiento geométrico

Los vértices pasan a ser transformados, según la lógica de nuestra aplicación. Estos vértices transformados son convertidos a nuestra superficie de dibujo, que puede

representar nuestra pantalla, impresora, o en memoria. Por último, convertiremos las unidades de nuestra superficie de dibujo a píxeles o puntos de impresión. Esto supone convertir nuestra escena vectorial o continua en nuestra imagen final y discreta en píxeles en la pantalla o como puntos de impresión.

Iremos agregando más etapas a nuestro procesamiento a medida que vayamos viendo más técnicas. Asimismo, cuando pasemos al modelado en tres dimensiones, agregaremos a nuestro procesamiento algunas etapas relacionadas con el modelado en tres dimensiones.

Referencia

Matrices de las Operaciones Fundamentales

Operación	**Matriz**	**Inversa**
Traslación: $T(d_x,d_y)$	$\mathbf{T} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ d_x & d_y & 1 \end{pmatrix}$	$\mathbf{T}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -d_x & -d_y & 1 \end{pmatrix}$
Cambio de escala: $S(s_x,s_y)$	$\mathbf{S} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\mathbf{S}^{-1} = \begin{pmatrix} 1/s_x & 0 & 0 \\ 0 & 1/s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$
Rotación: $R(\alpha)$	$\mathbf{R} = \begin{pmatrix} \cos\alpha & \operatorname{sen}\alpha & 0 \\ -\operatorname{sen}\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\mathbf{R}^{-1} = \begin{pmatrix} \cos\alpha & -\operatorname{sen}\alpha & 0 \\ \operatorname{sen}\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$
Sesgado en la X: $H_x(\alpha)$	$\mathbf{H}_x = \begin{pmatrix} 1 & 0 & 0 \\ \operatorname{cotg}\alpha & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\mathbf{H_x}^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ -\operatorname{cotg}\alpha & 1 & 0 \end{pmatrix}$

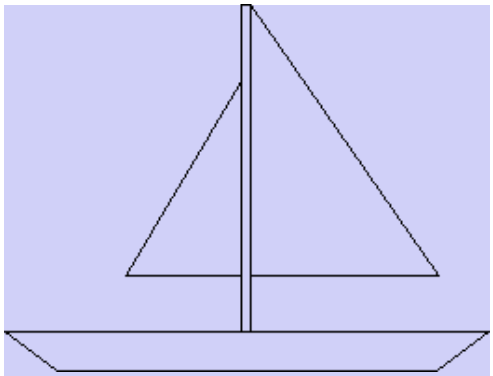
		$\begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$
Sesgado en la Y: $H_y(\alpha)$	$\mathbf{H_y} = \begin{pmatrix} 1 & \operatorname{cotg}\alpha & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\mathbf{H_y}^{-1} = \begin{pmatrix} 1 & -\operatorname{cotg}\alpha & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Ejercicios

Enlace al paquete *de este capítulo.*

1. Implemente una biblioteca para representar los puntos, vectores, y matrices al igual que sus operaciones, como pueden ser:
   - suma
   - resta
   - producto escalar
   - producto vectorial
   - módulo, para calcular la magnitud (o distancia) de un vector
   - normalizar un vector
   - calcular el ángulo entre dos vectores
   - determinante
   - inversa
   - traspuesta

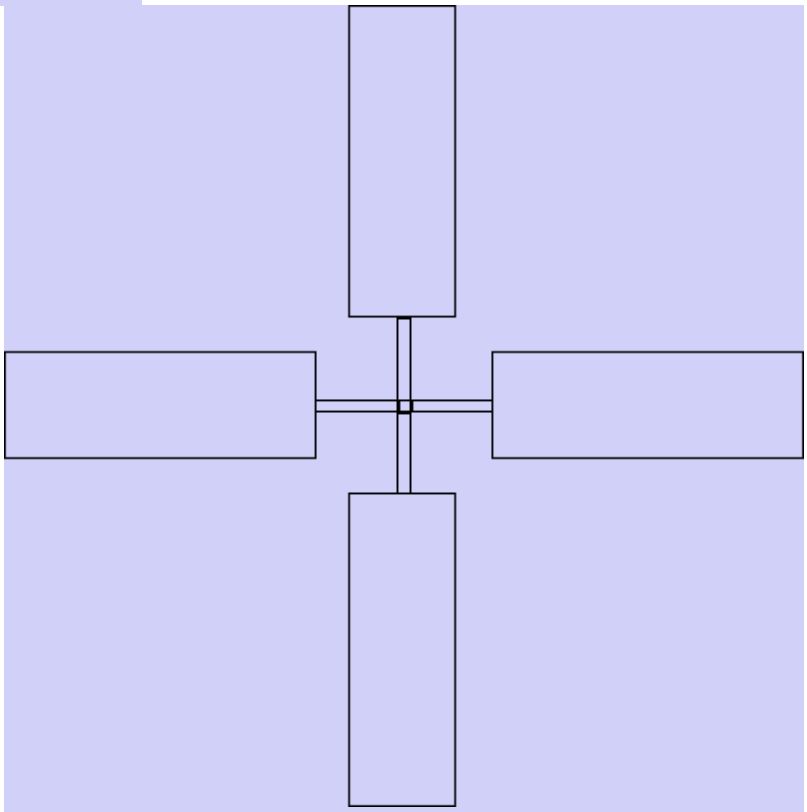
   Puede limitar la implementación de estas entidades y operaciones para representarlas y trabajar en coordenadas homogéneas.
2. Implemente las transformaciones afines, en coordenadas homogéneas: traslación, cambio de escala, rotación, y sesgado. Estas operaciones realmente construyen las matrices que luego serán multiplicadas.
3. Dibuje una figura compuesta que más le guste. Por ejemplo, puede modelar y dibujar la siguiente imagen de un velero, compuesto por un casco, un mástil, y dos velas.

4. 

Ejercicio 3 - Velero

Redibuje el ejemplo presentado en el curso del molino moderno en la figura 28 para que use unas aspas antiguas. Por ejemplo,

Es una buena idea modelar un aspa y luego crear cuatro instancias para ser situadas en sus lugares correctos para crear las cuatro aspas del molino.



Ejercicio 4 - Aspas de un molino

Capítulo 7 - Recorte

Siguiendo nuestro modelado en dos dimensiones, hablaremos de una operación importante acerca de líneas y polígonos. El recorte (o *clipping*, en inglés) modifica las líneas y polígonos a visualizar debidos a una región que los limita. Técnicamente, se trata de una intersección de dos entidades geométricas. Típicamente, la región de recorte es un rectángulo vertical, cuyos lados son paralelos a los ejes del sistema de coordenadas, que suele representar la vista de nuestra escena o la pantalla en sí.

Existen varios algoritmos y métodos para hacer el recorte. Por ahora, sólo miraremos aquellas técnicas analíticas. Esto significa que aplicaremos la operación de recorte a nuestro modelo en nuestro sistema de coordenadas matemático de nuestro mundo y escena. Existen otros métodos de recorte que se pueden aplicar en la etapa de conversión del modelo a píxeles. Trataremos estas técnicas en posteriores capítulos.

Veremos varios métodos para varios casos de recorte, como son el recorte de líneas y polígonos en un rectángulo vertical, al igual que en una región de recorte descrita por un polígono.

Recortando Líneas

Realmente, no pretendemos recortar líneas sino más bien segmentos. Como un segmento se describe por sus puntos extremos, no tenemos que consultar la infinidad de puntos que componen tal segmento.

De la misma manera que analizamos el recorte de puntos individuales, podemos aplicar la misma lógica para los puntos extremos de un segmento. El resultado es cuatro casos diferentes. Si ambos extremos están dentro del rectángulo de recorte, entonces no tenemos que hacer nada más que aceptar el segmento. Si un extremo está dentro y el otro fuera del rectángulo, entonces el segmento cruza uno de los lados de la región de recorte. Tenemos que calcular el punto de intersección lo cual modificará el extremo del segmento externo para que sea interno al rectángulo de recorte. Si ambos extremos están fuera del rectángulo de recorte, entonces el segmento puede estar completamente fuera del rectángulo, por lo que es rechazado inmediatamente. Sin embargo, tal segmento podría cruzar dos lados del rectángulo. En este caso, tenemos que realizar dos cálculos de intersección, modificando ambos extremos del segmento para que éste sea interior al rectángulo de recorte.

Veremos varios algoritmos para solucionar este problema: Algoritmo Exhaustivo, Algoritmo de Cohen-Sutherland, Algoritmo de Cyrus-Beck, Algoritmo de Liang-Barsky, y Algoritmo de Nicholl-Lee-Nicholl.

Recortando Polígonos

Lo normal es que nuestra escena contenga varios polígonos y no segmentos sueltos. Como nuestra vista limita el contenido de la escena que podemos mostrar, tendremos que recortar los polígonos de la escena para eliminarlos de la vista o dibujarlos parcial o completamente. Como representamos los polígonos mediante los segmentos que forman sus aristas, al recortar los polígonos en un rectángulo, acabaremos eliminando, aceptando, o recortando las aristas existentes y posiblemente añadiendo nuevas. Los diferentes resultados dependen del tipo de polígono a recortar.

Un polígono convexo que intersecta un rectángulo dará lugar a un polígono convexo. Un polígono cóncavo recortado por un rectángulo puede resultar en uno o varios polígonos cóncavos o convexos.

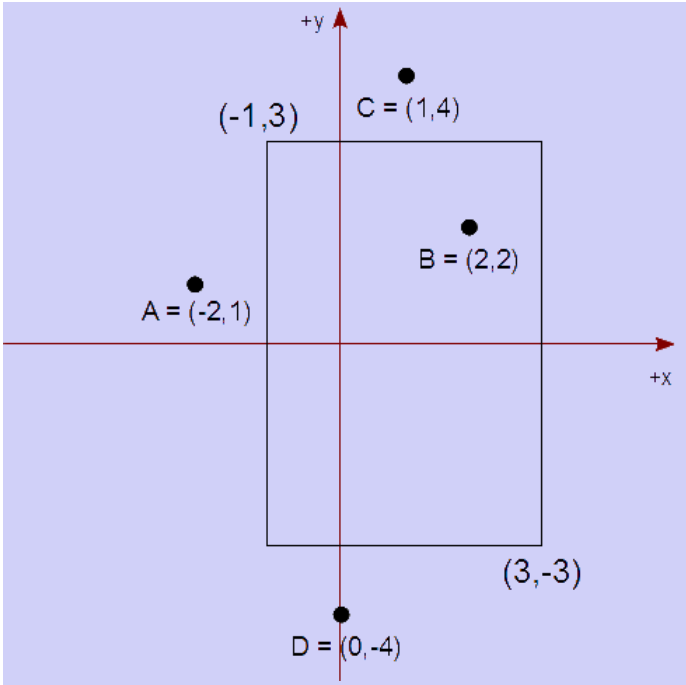
Veremos varios algoritmos para recortar polígonos: Algoritmo de Sutherland-Hodgman, Algoritmo de Liang-Barsky, y Algoritmo de Weiler-Atherton.

Recortando Puntos

Veamos el problema sencillo de recortar puntos. Para este caso, sólo necesitamos aceptar o rechazar tal punto. Nos basamos en los valores de las coordenadas del punto y de las del rectángulo vertical. Como el rectángulo está *de pie*, sólo tenemos que considerar las coordenadas de ciertos ejes y tratarlas como condiciones de contorno. Por consiguiente, el punto P = (x,y) debe satisfacer todas las siguientes inecuaciones para aceptarlo como un punto interior al rectángulo:

$$x_{izq} \leq x \leq x_{der}$$
$$y_{inf} \leq y \leq y_{sup}$$

donde, x_{izq}, x_{der}, y_{inf}, e y_{sup} son las coordenadas que describen los lados del rectángulo vertical de recorte izquierdo, derecho, inferior, y superior, respectivamente. Esto lo podemos ver fácilmente en las figuras 1 y 2:



Figuras 1 - Antes de Recortar Puntos

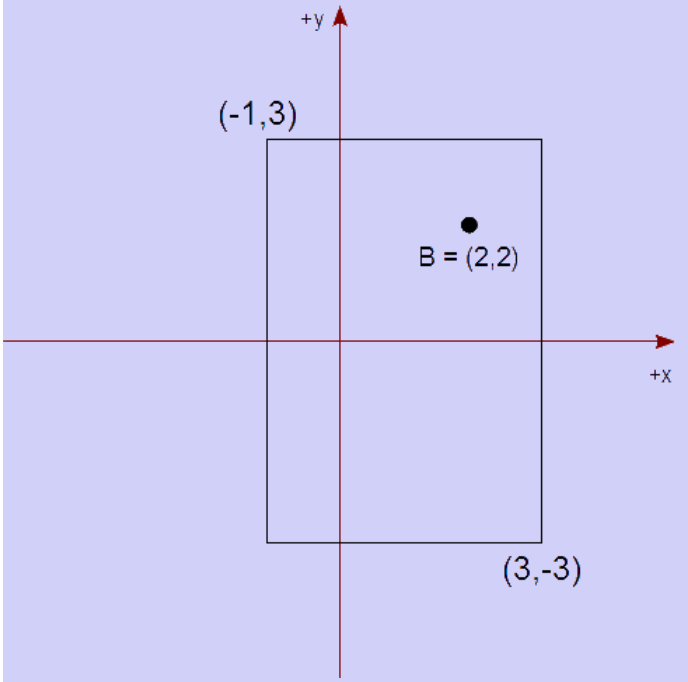


Figura 2 - Puntos Recortados

Algoritmo Exhaustivo

Si no podemos aceptar inmediatamente un segmento, entonces comprobamos tal segmento con cada arista del rectángulo. Cada comprobación da lugar a una intersección con un lado del rectángulo, acortando el segmento hasta que esté dentro de ello. A continuación, comprobamos cada intersección es interior a nuestro rectángulo de recorte.

Con esta solución, debemos resolver varias ecuaciones por cada intersección entre segmento y arista del rectángulo. Podríamos usar la fórmula de la intersección de la pendiente, pero ésta sirve para líneas (infinitas) y no para segmentos. Además, esta fórmula no puede describir líneas verticales. Sin embargo, podemos usar una forma paramétrica para describir cualquier punto del segmento con los extremos (x_0,y_0) y (x_1,y_1):

$$x = x_0 + t\ (x_1 - x_0)$$
$$y = y_0 + t\ (y_1 - y_0)$$

donde, *t* queda comprendido en el intervalo [0,1].

También debemos describir cada arista del rectángulo de la misma forma con los extremos (x'_0,y'_0) y (x'_1,y'_1):

$$x' = x'_0 + t'\ (x'_1 - x'_0)$$
$$y' = y'_0 + t'\ (y'_1 - y'_0)$$

donde, *t'* queda comprendido en el intervalo [0,1].

Nos interesa el punto de intersección del segmento con cada arista del rectángulo de recorte. Esto significa que debemos encontrar el punto común para ambos segmentos, por lo que (x,y) = (x',y'). Como sabemos la coordenada de intersección para un arista concreta del rectángulo vertical, el cálculo de *t* es sencillo. Sólo necesitamos calcular la otra coordenada basándonos en

tal valor de *t*. Por ejemplo, si queremos determinar la intersección de un segmento con el arista x_{izq} del rectángulo de recorte, entonces sabemos la coordenada del punto de intersección $x = x_{izq}$. Por lo tanto, tendremos que hallar la coordenada *y* a través del parámetro *t*, que calcularíamos previamente.

Ahora bien, ambos parámetros *t* - calculado del segmento - y *t'* de la arista deben yacer en el intervalo [0,1], para considerar el punto (x,y) como interior al rectángulo de recorte. Por consiguiente, aceptamos tal punto de intersección como válido para el recorte del segmento. También debemos considerar el caso de un segmento paralelo a una arista, antes de resolver las ecuaciones.

Concluimos que este método involucra varios cálculos y comprobaciones y que por tanto se considera ineficiente.

Ejemplo

Veamos un ejemplo sencillo para entender este método.

Descripción

Tenemos un rectángulo de recorte cuya diagonal es de (-1,3) a (3,-3) que representa nuestra vista. Queremos mostrar el segmento AB, en tal rectángulo de recorte, descrito por los puntos,

$A = (-2, 1)$ y $B = (2, 2)$

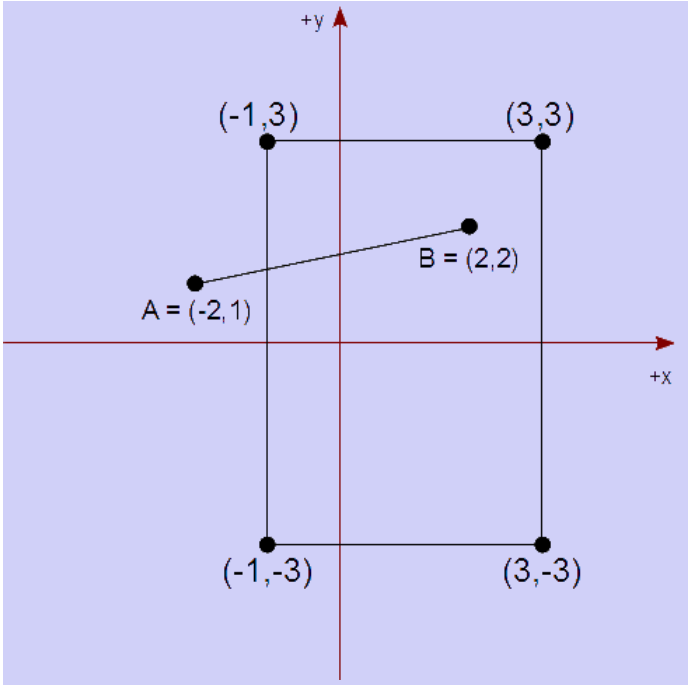


Figura 3 - Ejemplo del Método Exhaustivo

Solución

Comenzamos estableciendo las ecuaciones paramétricas de la línea que contiene el segmento AB y la que contiene la arista superior PQ del rectángulo de recorte,

$$\{ x = A_x + t\ (B_x - A_x)$$
$$\{ y = A_y + t\ (B_y - A_y)$$

$$\{ x' = P_x + t' (Q_x - P_x)$$
$$\{ y' = P_y + t' (Q_y - P_y)$$

Esto quedaría así:

$$\{ x = -2 + 4 t$$
$$\{ y = 1 + t$$

$$\{ x' = -1 + 4 t'$$
$$\{ y' = 3 + 0 t'$$

Como (x,y) = (x′,y′), calculamos *t* y *t*′:

$$\{ -2 + 4 t = -1 + 4 t'$$
$$\{ 1 + t = 3$$

Dando,

$$t = 3 - 1 = 2$$
$$t' = 7 / 4 = 1,75$$

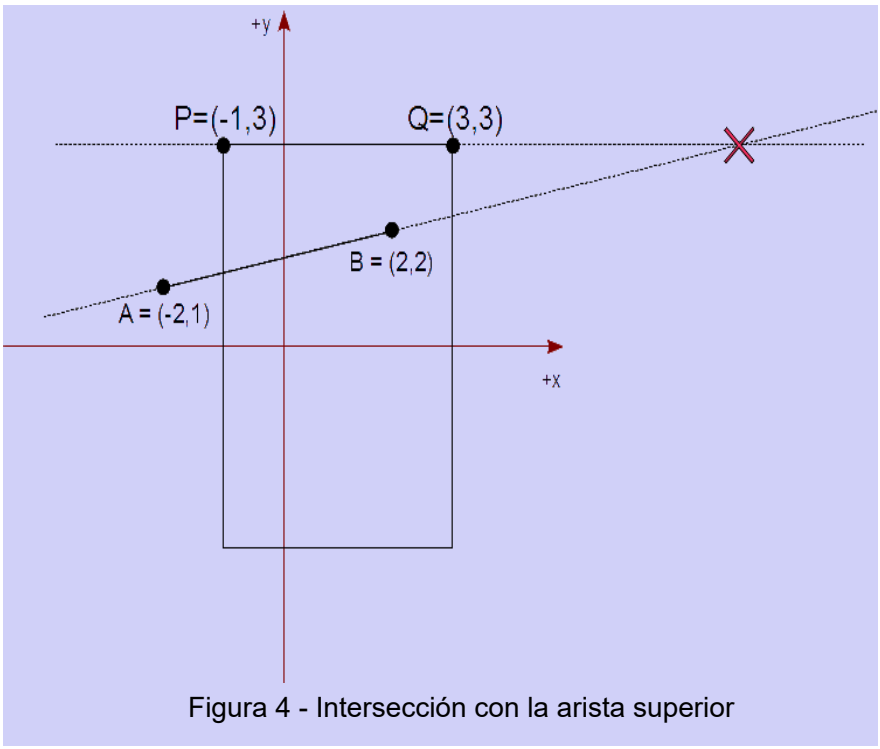


Figura 4 - Intersección con la arista superior

Como ninguno de estos valores queda comprendido entre 0 y 1, no calculamos el punto de intersección porque no pertenece a ambos segmentos AB ni PQ.

Establecemos las ecuaciones paramétricas de la línea que contiene el segmento AB y la que contiene la arista derecha QR del rectángulo de recorte,

$$\{ x' = Q_x + t' (R_x - Q_x)$$
$$\{ y' = Q_y + t' (R_y - Q_y)$$

Esto quedaría así:

$$\{ x = -2 + 4 t$$
$$\{ y = 1 + t$$

$$\{ x' = 3 + 0 t'$$
$$\{ y' = 3 - 6 t'$$

Como (x,y) = (x′,y′), calculamos *t* y *t*′:

$$\{ -2 + 4 t = 3$$
$$\{ 1 + t = 3 - 6 t'$$

Dando,

$$t = 5 / 4 = 1,25$$
$$t' = 1 / 8 = 0,125$$

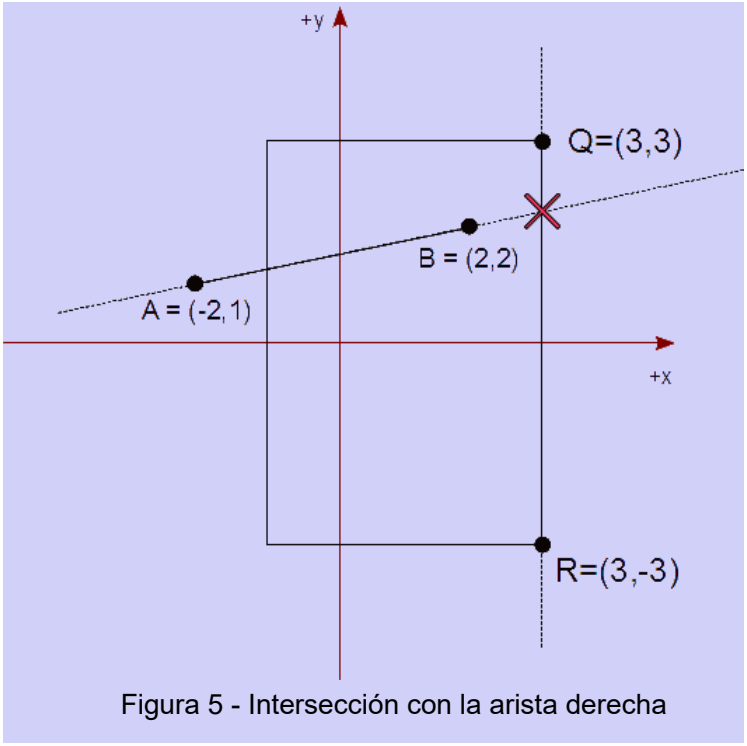


Figura 5 - Intersección con la arista derecha

Aunque *t′* queda comprendido entre 0 y 1, *t* no, por lo que no calculamos el punto de intersección porque no pertenece a ambos segmentos AB ni QR.

Establecemos las ecuaciones paramétricas de la línea que contiene el segmento AB y la que contiene la arista inferior RS del rectángulo de recorte,

$$\begin{cases} x' = R_x + t' (S_x - R_x) \\ y' = R_y + t' (S_y - R_y) \end{cases}$$

Esto quedaría así:

$$\begin{cases} x = -2 + 4\ t \\ y = 1 + t \end{cases}$$

$$\begin{cases} x' = 3 - 4\ t' \\ y' = -3 + 0\ t' \end{cases}$$

Como (x,y) = (x′,y′), calculamos *t* y *t′*:

$$\begin{cases} -2 + 4\ t = 3 - 4\ t' \\ 1 + t = -3 \end{cases}$$

Dando,

$$t = -4$$
$$t' = 21 / 4 = 5{,}25$$

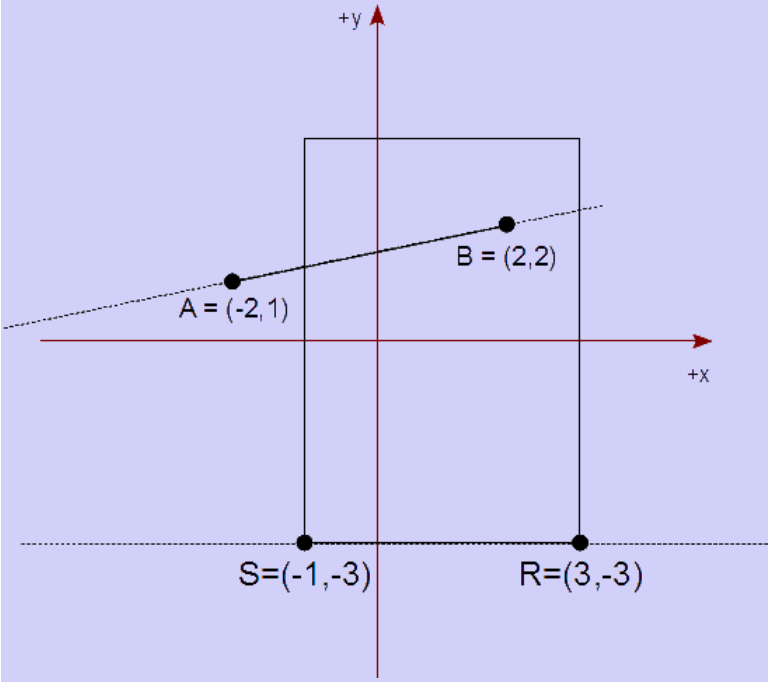


Figura 6 - Intersección con la arista inferior

Obviamente ninguno de estos valores paramétricos queda comprendido entre 0 y 1, por lo que no calculamos el punto de intersección porque no pertenece a ambos segmentos AB ni RS.

Establecemos las ecuaciones paramétricas de la línea que contiene el segmento AB y la que contiene la arista izquierda SP del rectángulo de recorte,

$$\begin{cases} x' = S_x + t' (P_x - S_x) \\ y' = S_y + t' (P_y - S_y) \end{cases}$$

Esto quedaría así:

$$\begin{cases} x = -2 + 4\ t \\ y = 1 + t \end{cases}$$

$$\begin{cases} x' = -1 + 0\ t' \\ y' = -3 + 9\ t' \end{cases}$$

Como (x,y) = (x′,y′), calculamos *t* y *t′*:

$$\begin{cases} -2 + 4\ t = -1 \\ 1 + t = -3 + 9\ t' \end{cases}$$

Dando,

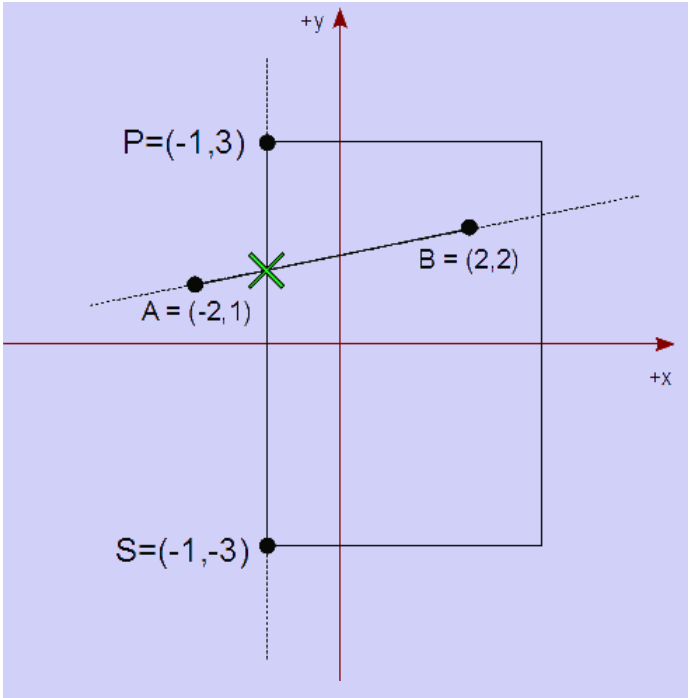


Figura 7 - Intersección con la arista izquierda

```
t  = 1 / 4 = 0,25
t' = 4,25 / 9 ≅ 0,4722
```

Ambos valores paramétricos quedan comprendidos entre 0 y 1, por lo que calculamos el punto de intersección a partir de cualquiera de los dos segmentos. Por ejemplo, usando el segmento AB, volvemos a las ecuaciones paramétricas con el parámetro t = 0,25,

```
{ x  = -2 + 4 (0,25) = -1
{ y  =  1 +   (0,25) =  1,25
```

Dando el punto de intersección (-1, 1,25).

Vemos que debemos acortar el segmento AB dando lugar a nuevos valores usando este punto de intersección:

```
A' = (-1, 1,25)
B  = ( 2, 2)
```

Al final, el segmento AB es acortado al segmento A′B.

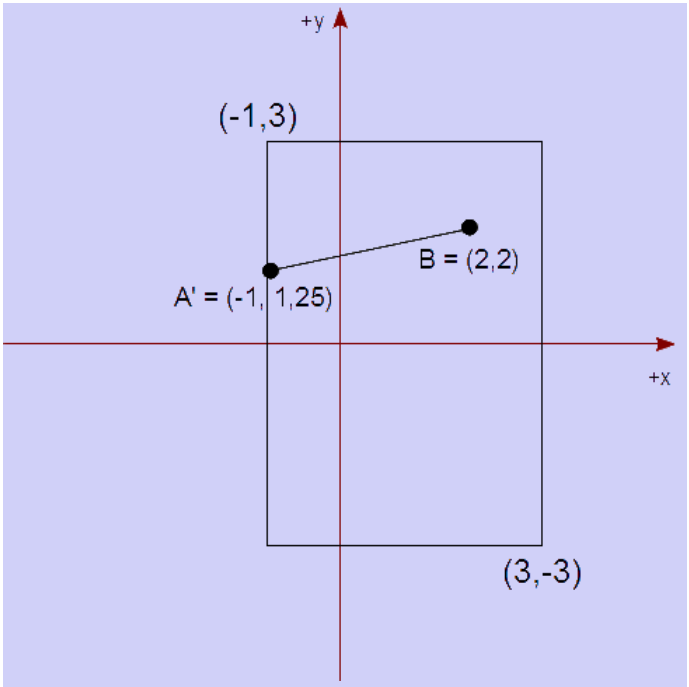


Figura 8 - Ejemplo Solución

Algoritmo de Cohen-Sutherland

Este algoritmo realiza varias comprobaciones iniciales para descubrir si se puede evitar cálculos de las intersecciones. El primer paso es comprobar si los puntos extremos del segmento son aceptados por sus posiciones. Si esto no es posible, entonces realizamos varias comprobaciones por regiones exteriores al rectángulo de recorte, formadas por las líneas rectas al extender sus aristas. Asimismo, podemos rechazar un segmento inmediatamente si sus extremos yacen en regiones a la izquierda de x_{izq}, por encima de y_{sup}, a la derecha de x_{der}, y por debajo de y_{inf}.

Si el segmento no se puede aceptar ni rechazar inmediatamente, entonces es partido en dos segmentos por un arista del rectángulo de recorte, para que uno de ellos sea rechazado de inmediato. Esto implica que implementamos un método iterativo de recorte para el segmento hasta que éste sea aceptado o rechazado. Este algoritmo es eficiente especialmente en el caso de tener un rectángulo de recorte muy grande que comprende a todos o una gran mayoría de los segmentos a visualizar. Asimismo, podemos tener el caso de un rectángulo de recorte muy pequeño que fuerza a todos o casi todos los segmentos a ser rechazados.

Se le asigna un código de cuatro bits, $b_3b_2b_1b_0$, a cada región de las nueve creadas. Este código binario es creado estratégicamente para representar rápida y fácilmente cada región. El valor de 1 indicará que un extremo yace en tal región mientras que un 0 indicará lo contrario.Aquí tenemos la estrategia a usar:

Bit	Condición	Descripción
b_3	$y > y_{sup}$	por encima de la arista superior
b_2	$y < y_{inf}$	por debajo de la arista inferior
b_1	$x > x_{der}$	a la derecha de la arista derecha
b_0	$x < x_{izq}$	a la izquierda de la arista izquierda

Usaremos ciertas operaciones a nivel de bit para asginar el código regional a cada extremo en el que yace y también para determinar tal región cuando necesitemos discriminarlos. Una forma eficiente de asignar un valor a un bit particular es tomando el primer bit que indica el signo positivo o negativo de la resta entre las dos coordenadas de la condición. Esto es, b_3 es asignado el bit del signo de (y_{sup}-y), b_2 es asignado el bit de (y - y_{inf}), b_1 es asignado el bit de (x_{sup}-x), y b_0 es asignado el bit de (x - x_{inf}).

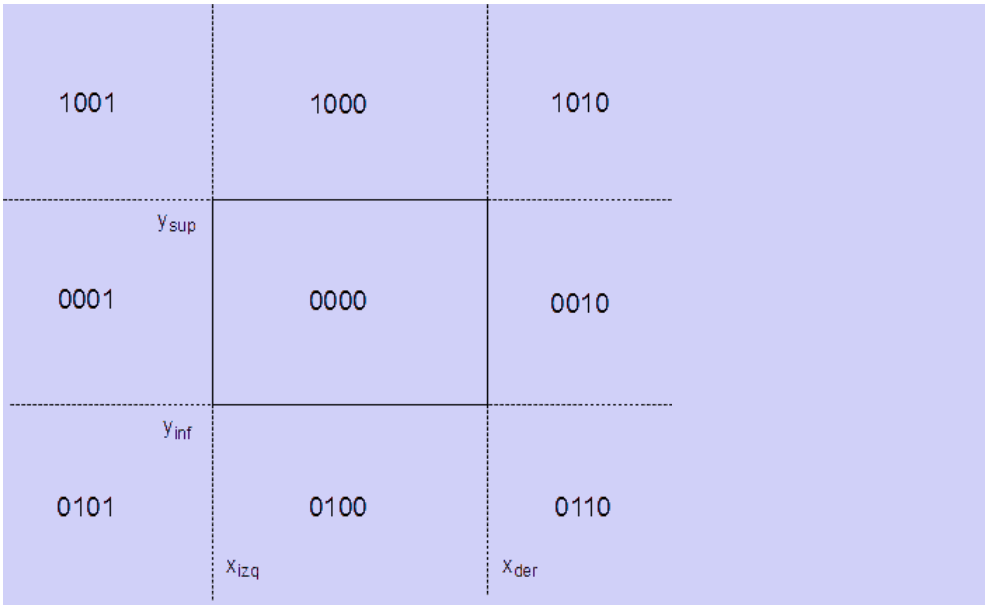


Figura 9 - Las regiones con los Códigos del Algoritmo de Cohen-Sutherland

La figura 9 muestra las regiones con sus correspondientes códigos.

Con estos códigos podemos determinar si los extremos, y por tanto el segmento que representan, pueden ser aceptados o rechazados inmediatamente. Obviamente, si ambos códigos son 0, entonces ambos extremos existen dentro del rectángulo de recorte, y por consiguiente el segmento es aceptado de inmediato. Para combinar estos dos códigos, simplemente aplicamos una operación **OR** y comprobamos si el resultado es 0. Si el código no es 0, entonces aplicamos la operación **AND** a ambos códigos. Si el resultado no es cero, entonces rechazamos el segmento de inmediato.

Si no podemos aceptar ni rechazar inmediatamente el segmento, entonces iremos dividiendo el segmento y comprobando sus nuevos códigos, para determinar su aceptación o rechazo. Dividimos el segmento calculando la intersección con las líneas creadas al extender las aristas del rectángulo de recorte. Si no podemos discriminar el nuevo segmento, entonces debemos continuar dividiéndolo hasta que al final lo aceptamos o lo rechazamos. Una propiedad notable es que modificamos el segmento *mudando* el extremo, cuyo código no es cero. Dicho de otra manera, el algoritmo elige un extremo exterior que será *mudado* a la intersección con la arista o con la línea que separa y define las regiones. Asignamos el código regional del punto de intersección al extremo *mudado*.

La desventaja de este algoritmo es que puede realizar varias intersecciones antes de alcanzar la intersección final y válida. Una mejora es calcular la pendiente para determinar el punto de intersección una sola vez y conservarla para posteriores iteraciones. Aun con esta mejora, este algoritmo no es tan eficiente. Por otro lado, la ventaja de este algoritmo es su simplicidad y además se puede extender fácilmente al caso de un volumen de recorte en 3D.

Ejemplo

Descripción

Queremos mostrar varios segmentos, pero únicamente dentro de nuestra vista representada por un rectángulo vertical descrito con las siguientes esquinas: superior izquierda (-1,3) e inferior derecha (3,-3). Tenemos los siguientes segmentos definidos por parejas de puntos que forman sus extremos:
A = (-2, 1) y B = (2, 2),
C = (1, 4) y D = (0,-4),
E = (4, 3) y F = (3, 0),
G = (-3,-1) y H = (-2,-4).

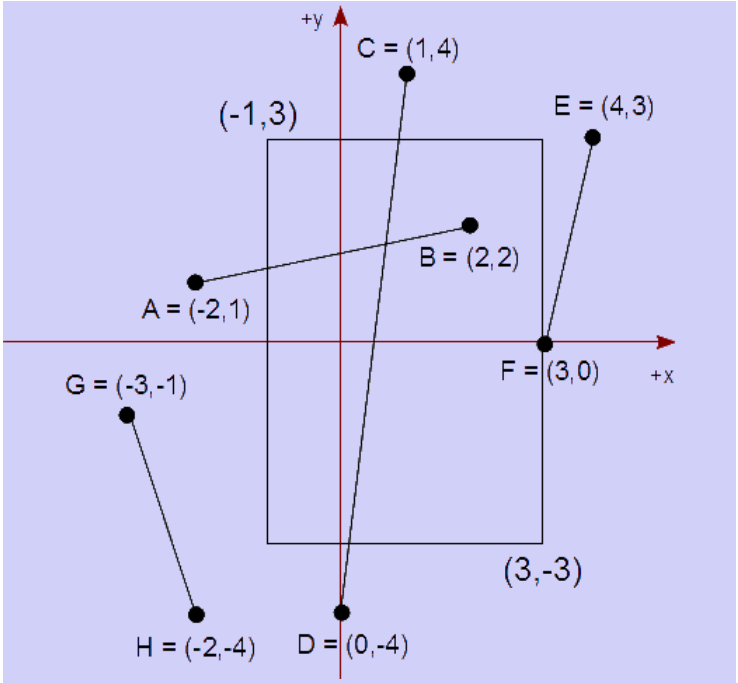


Figura 10 - Ejemplo del Algoritmo de Cohen-Sutherland

Solución

Calculamos los códigos regionales para cada punto:

```
A: 0001  B: 0000
C: 1000  D: 0100
E: 0010  F: 0000
G: 0001  H: 0101
```

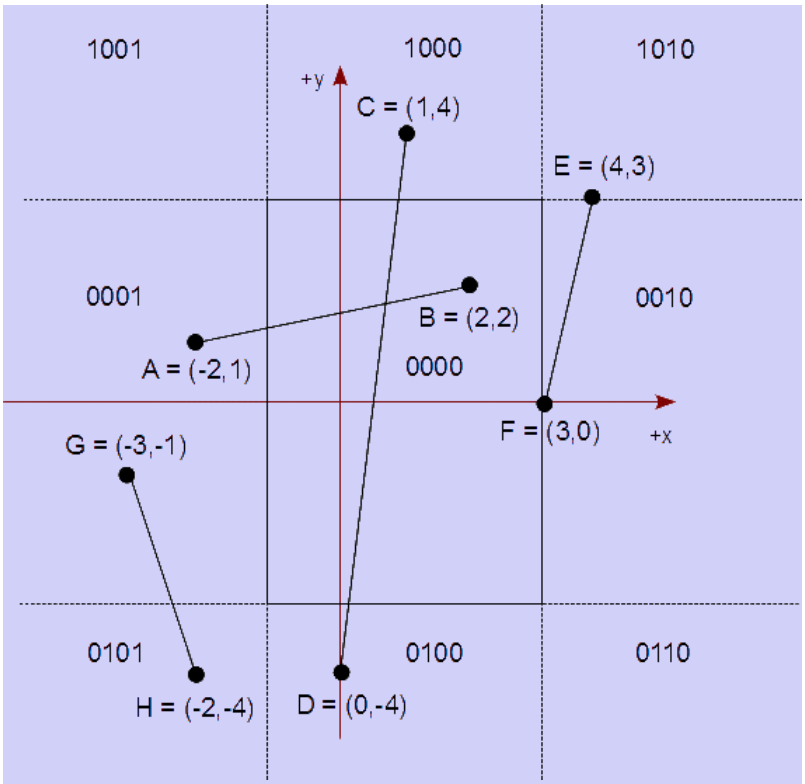


Figura 11 - Ejemplo Solución con Códigos

Aplicamos nuestros criterios a los resultados de las operaciones a nivel de bit de los códigos regionales de cada punto dando lugar a:

```
AB: 0001 OR 0000 = 0001  ⇒
0001 AND 0000 = 0000  ⇒  Hay
que recortar
CD: 1000 OR 0100 = 1100  ⇒
1000 AND 0100 = 0000  ⇒  Hay
que recortar
EF: 0010 OR 0000 = 0010  ⇒
0010 AND 0000 = 0000  ⇒  Hay
que recortar
GH: 0001 OR 0101 = 0101  ⇒
0001 AND 0101 = 0001  ⇒
Inmediatamente lo rechazamos
```

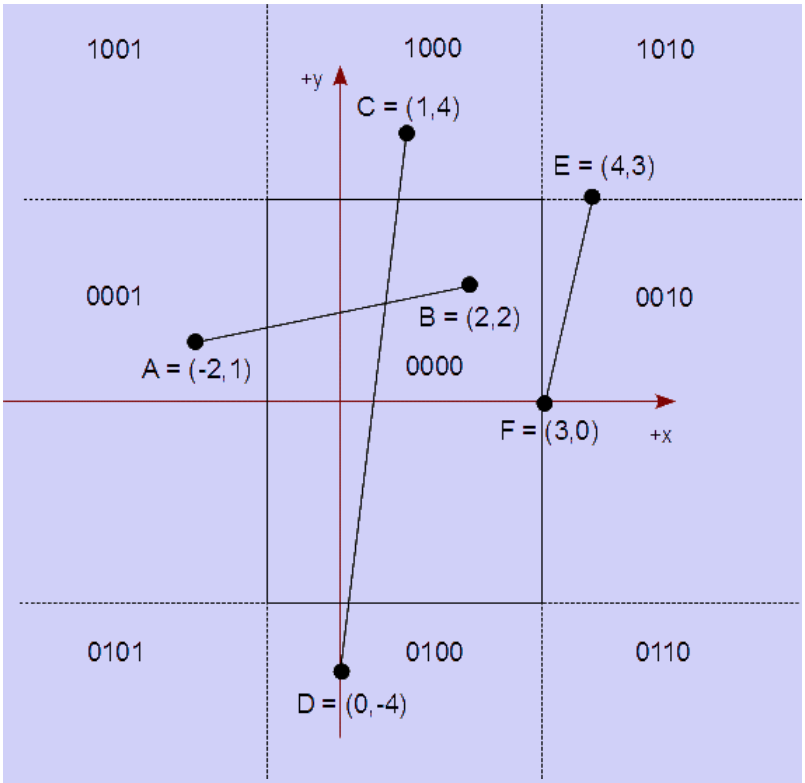


Figura 12 - Ejemplo Solución Aplicando Criterios

Hacemos la primera

tanda de recortes de los puntos con sus primeras aristas del algoritmo, obteniendo:

```
{ A'x = -1
{ A'y =  1 + 1 * 1 / 4 = 1,25

{ C'x =  1 - 1 * 1 / (-8) = 1,125
{ C'y =  3

{ E'x =  3
{ E'y =  3 - 3 * (-1) / (-1) = 0
```

Recalculamos los códigos de cada punto:

```
A'B: 0000 OR 0000 = 0000 ⇒ Aceptamos el segmento
C'D: 0000 OR 0100 = 0100 ⇒ 0000 AND 0100 = 0000 ⇒ Hay que recortar
E'F: 0000 OR 0000 = 0000 ⇒ Aceptamos el segmento
```

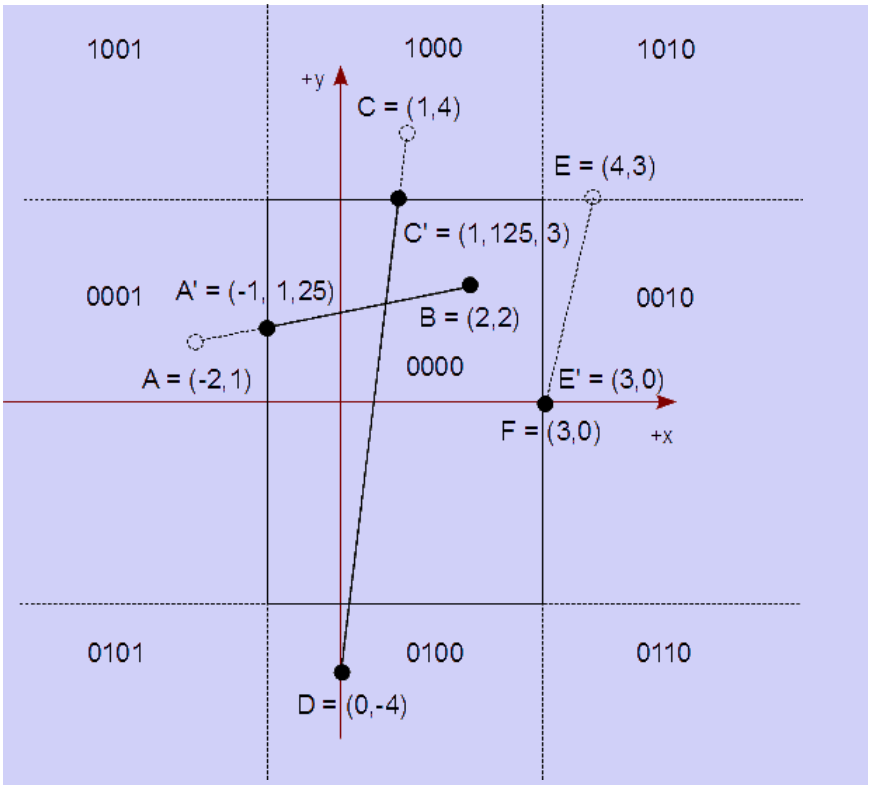


Figura 13 - Ejemplo Solución Recortando con las Primeras Aristas

Nos falta recortar una segunda vez para el segmento C'D. El algoritmo elige el punto D para recortar, obteniendo:

```
{ D'x =  0 + 1,125 * 1 / 7 = 0,1607
{ D'y = -3
```

Recalculamos los códigos de cada punto:

```
C'D': 0000 OR 0000 = 0000 ⇒ Aceptamos el segmento
```

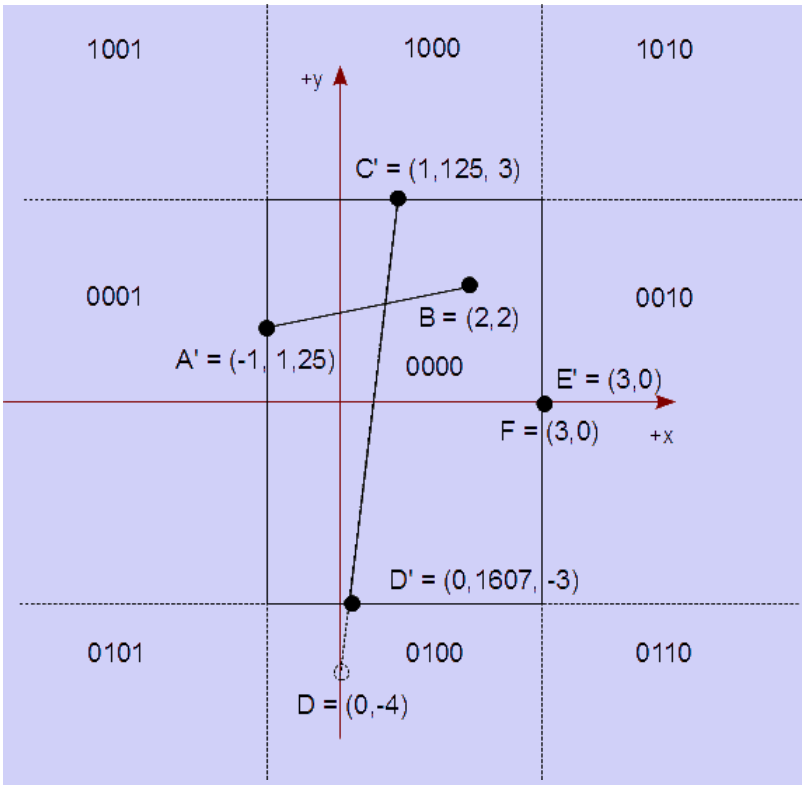


Figura 14 - Ejemplo Solución Recortar C'D

Al final, obtenemos la siguiente imagen con los segmentos recortados e interiores al rectángulo de recorte.

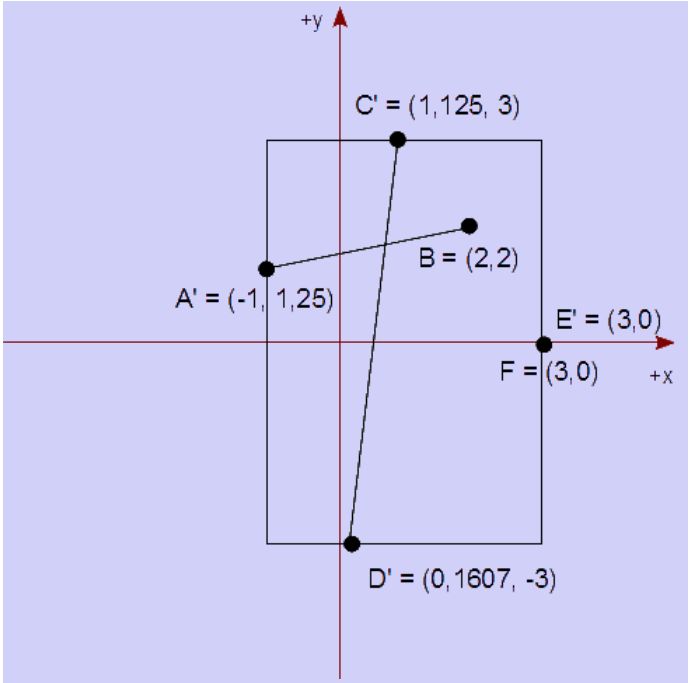


Figura 15 - Ejemplo Solución Final

Algoritmo

La función principal es *Recortar()*, la cual aceptará tres parámetros: dos puntos extremos, P y Q, donde cada uno contiene las coordenadas (x,y) que representan el segmento a recortar, y un rectángulo, R, que contiene los elementos $\{x_{izq}, y_{sup}, x_{der}, y_{inf}\}$ representando las coordenadas de las dos esquinas izquierda superior y derecha inferior. También debemos definir los códigos binarios para: **SUPERIOR**=8, **INFERIOR**=4, **DERECHA**=2, e **IZQUIERDA**=1.

Para *Recortar()*, el algoritmo es:

```
booleano Recortar( ref Punto P, ref Punto Q, Rectángulo R )
        1. aceptar ← falso
        2. terminar ← falso
        3. códigoP ← Calcular_Código(P,R)
        4. códigoQ ← Calcular_Código(Q,R)
        5. Repetir:
                6. Si (códigoP OR códigoQ) = 0, entonces
                        7. aceptar ← verdadero
                        8. terminar ← verdadero
                9. Si no, compruebe que  (códigoP AND códigoQ) ≠ 0,
                        10. terminar ← verdadero
                11. Si no, entonces
                        12. Si códigoP = 0, entonces
                                13. código ← códigoQ
                        14. Si no, entonces
                                15. código ← códigoP
                        16. Si código es SUPERIOR, entonces
                                17. Δy ← Q.y-P.y
                                18. Si Δy = 0, entonces  // segmento vertical
                                        19. x ← P.x
                                20. Si no, entonces
                                        21. x ← P.x + (Q.x-P.x)*(R.ysup-P.y) / Δy
                                22. y ← ysup
```

```
                        23. Si no, compruebe que código es INFERIOR
                                24. Δy ← Q.y-P.y
                                25. Si Δy = 0, entonces  // segmento vertical
                                        26. x ← P.x
                                27. Si no, entonces
                                        28. x ← P.x + (Q.x-P.x)*(R.y_inf-P.y) / Δy
                                29. y ← y_inf
                        30. Si no, compruebe que código es DERECHA
                                31. x ← x_der
                                32. y ← P.y + (Q.y-P.y)*(R.x_der-P.x) / (Q.x-P.x)
                        33. Si no, entonces  // código = IZQUIERDA
                                34. x ← x_izq
                                35. y ← P.y + (Q.y-P.y)*(R.x_izq-P.x) / (Q.x-P.x)
                        36. Si código = códigoP, entonces
                                37. P.x ← x
                                38. P.y ← y
                                39. códigoP ← Calcular_Código(P,R)
                        40. Si no, entonces
                                41. Q.x ← x
                                42. Q.y ← y
                                43. códigoQ ← Calcular_Código(Q,R)
        44. Mientras que terminar = falso
        45. Terminar( aceptar )
```

Si el segmento es aceptado, entonces el algoritmo terminará con el valor booleano *verdadero*. Además, los extremos, P y Q, contendrán los valores recortados por este mismo algoritmo. Si el segmento es rechazado, entonces uno debería hacer caso omiso de los parámetros, P y Q. En cualquier caso, se pasan estos dos parámetros por referencia, para así poder modificarlos.

A continuación presentamos el algoritmo para *Calcular_Código()*, el cual requiere un punto, P, y el rectángulo de recorte, R:

```
entero Calcular_Código( Punto P, Rectángulo R )
        1. código ← 0
        2. Si P.y > R.y_sup, entonces
                3. código ← código OR SUPERIOR
        4. Si no, compruebe que P.y < R.y_inf
                5. código ← código OR INFERIOR
        6. Si P.x > R.x_der, entonces
                7. código ← código OR DERECHA
        8. Si no, compruebe que P.x < R.x_izq
                9. código ← código OR IZQUIERDA
        10. Terminar( código )
```

No todos los lenguajes de programación aceptan realizar operaciones a nivel de bit con números que no sean enteros. Tampoco todos los lenguajes establecen exactamente la cantidad de bits para representar internamente un número real. Por estas razones, no hemos presentado el algoritmo mejorado para *Calcular_Código()* que previamente hablamos en este apartado.

Suponiendo que un número real se representase con 32 bits y el 32º bit contiene su signo positivo, como 0, o negativo, como 1, aquí presentamos otra versión del algoritmo *Calcular_Código()*:

```
entero Calcular_Código( Punto P, Rectángulo R )
        1. código ← (((P.y - R.y_sup) SHR 31) SHL 3) OR
                (((R.y_inf - P.y) SHR 31) SHL 2) OR
        (((P.x - R.x_der) SHR 31) SHL 1) OR
```

```
        ((R.x_izq - P.x)  SHR 31)
        2. Terminar( código )
```

El operador *SHL* se refiere a la operación común de desplazamiento de bits a la izquierda y el operador *SHR* se refiere a la del desplazamiento de bits a la derecha.

Algoritmo de Cyrus-Beck

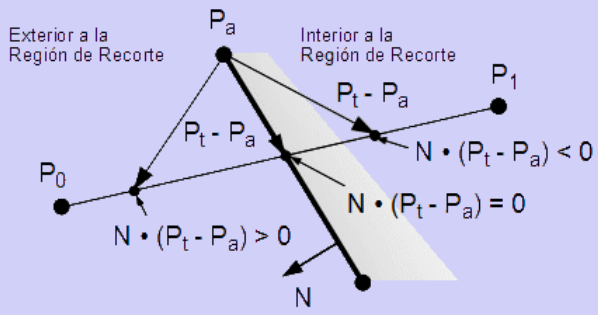


Figura 16 - Tres ejemplos de puntos fuera, dentro, y en la arista de la región de recorte

Este algoritmo se basa en calcular las intersecciones del segmento, P_0P_1, con cada arista de un polígono convexo de recorte. Se usa la ecuación paramétrica de la línea para determinar los cuatro valores de *t*. Posteriormente, realizamos varias comparaciones para determinar cuáles de estos cuatro valores de *t* corresponden a intersecciones reales. Sólo al tener los valores válidos de *t* es cuando calculamos los puntos de intersección. Para encontrar el punto de intersección, el algoritmo de Cyrus-Beck se basa en la ecuación paramétrica de una línea recta. Como tenemos dos segmentos: P_0P_1 y la arista, que yacen sobre dos líneas rectas, tenemos dos ecuaciones paramétricas que las representan. Al resolver estas dos ecuaciones, averiguaremos un valor de *t* con el que determinaremos el punto de intersección. La ecuación paramétrica es la siguiente:

$P(t) = P_0 + (P_1-P_0)\, t$

donde *t* queda comprendido en el intervalo [0,1]. Esto implica que $P(0) = P_0$ y $P(1) = P_1$.

Para determinar el valor exacto de *t*, haremos uso del producto escalar entre dos vectores. El primer vector será el vector normal de la arista que *apunta* hacia fuera de la región de recorte. El segundo vector será uno que coincide con la arista. Crearemos este segundo vector a partir de un punto cualquiera, P_a, en la arista, y el punto P(t) que es común a la línea del segmento, P_0P_1, que queremos recortar y a esta misma arista. El cálculo es el siguiente:

$\mathbf{N}\cdot(P(t) - P_a)$

Si el producto escalar es positivo, entonces el punto, P(t), yace fuera de la región de recorte y si es negativo, entonces P(t) yace dentro. Como nos interesa averiguar el punto común que yace en la arista y por tanto el borde de la región de recorte, el producto escalar debe ser 0; véase la figura 16. Esto significa que debemos resolver la siguiente ecuación:

$\mathbf{N}\cdot(P(t) - P_a) = 0$

Sustituimos P(t) por su definición,

$\mathbf{N}\cdot(P_0 + (P_1-P_0)\, t - P_a) = 0$

Aplicamos la propiedad distributiva,

$\mathbf{N}\cdot(P_0 - P_a) + \mathbf{N}\cdot(P_1-P_0)\, t = 0$

Dejemos que $\mathbf{D} = P_1-P_0$, que es el vector de P_0 a P_1 del segmento. Ahora resolvemos para *t*:

```
N· (P0  -  Pa)
```

$$t = \frac{}{-\mathbf{N}\cdot\mathbf{D}}$$

Este valor de *t* es válido solamente si el denominador no es cero. Esto significa que las siguientes condiciones deben mantenerse:

N ≠ 0 El vector normal jamás debería ser nulo. Si ocurriere lo contrario, entonces se trata de un error.

D ≠ 0 Esto significa que $P_0 \neq P_1$. Si ocurriere lo contrario, entonces se trata de un solo punto y no de un segmento.

N·D ≠ 0 Esto significa que la arista y el segmento no son paralelos. Si fueren paralelos, entonces no existe ninguna intersección.

El algoritmo calcula cada intersección entre el mismo segmento y cada arista de la región de recorte. Para este cálculo, determinamos el vector normal de cada arista y un punto arbitrario que yace en una arista. Lo más sencillo y práctico es elegir un punto conocido como un vértice que forma la arista. Posteriormente, debemos elegir cuáles de los cuatro valores de *t* corresponden a intersecciones internas a la región de recorte. Primeramente, comprobamos que los valores de *t* quedan comprendidos en el intervalo [0,1]. Si no se cumple esta condición, entonces significa que el punto de intersección no yace en el segmento y por tanto descartamos tal valor de *t*. Como cada cálculo que hacemos se basa en encontrar las intersecciones entre dos líneas rectas, esto significa que un valor de *t* puede indicar una intersección que no yace en una arista.

En algunos casos, no es tan sencillo determinar si una intersección yace dentro o fuera de la región de recorte. Podríamos realizar varias comprobaciones, pero entonces ralentizaríamos el algoritmo y no conseguiríamos una ventaja sobre otros algoritmos, como el de Cohen-Sutherland. Analizando varios casos de intersecciones entre segmentos y aristas, descubrimos un comportamiento que nos servirá para discriminar tal intersección y así determinar si es aceptada o rechazada. Esta técnica se basa en denominar las intersecciones como potencialmente entrantes (PE) y potencialmente salientes (PS) de la región de recorte. Si cruzamos una arista, yendo desde los extremos P_0 a P_1, que nos hace entrar en la mitad del plano, entonces la intersección lleva la etiqueta de PE. Si por el contrario, al cruzar la arista, nos hace salir del plano interior creado por la arista, entonces tal intersección es PS. Con esta clasificación, notamos que dos intersecciones interiores a la región de recorte tienen etiquetas opuestas.

Para clasificar si una intersección es PE o PS, podemos basarnos en el ángulo formado por los vectores del segmento P_0P_1 y el normal de una arista. Si el ángulo es mayor de 90°, entonces es PE; y si es menor de 90°, entonces es PS. No requerimos calcular el ángulo, ya que esta información está contenida en el producto escalar del vector normal **N** y el vector **D** del segmento P_0P_1:

N·D < 0 ⇒ ángulo > 90° ⇒ PE

N·D > 0 ⇒ ángulo < 90° ⇒ PS

Vemos que **N·D** es el denominador en el cálculo de *t* para determinar el punto de intersección. Por lo tanto, podemos clasificar el valor de *t* mientras lo calculamos.

Nos interesa encontrar aquellos puntos de intersección etiquetados de PE a PS. El punto de intersección con la etiqueta PE que sea interior a la región de recorte es aquél que tenga el mayor valor de *t*; lo llamaremos, t_e. El punto interior que tenga la etiqueta PS es aquél que tenga el menor valor de *t*, al cual lo llamaremos, t_s. El segmento cruzante se define en el intervalo $[t_e,t_s]$. Como ya mencionamos anteriormente, los valores de *t* deben estar comprendidos en el intervalo de [0,1]. Por consiguiente, la cota inferior de $t_e = 0$ y la cota superior de $t_s = 1$. Si $t_e > t_s$, entonces el segmento no es interior a la región de recorte y por tanto lo rechazamos. Al final, obtendremos los valores correctos de t_e y t_s para así calcular las coordenadas *x* e *y* de los puntos de intersección.

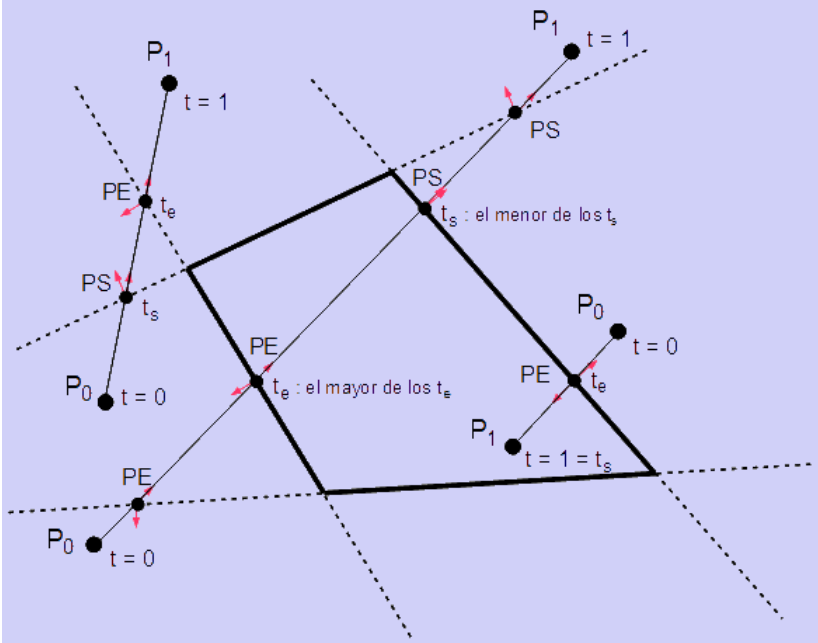


Figura 17 - Tres casos de segmentos cruzando las aristas del polígono de recorte con las etiquetas PE y PS

Este algoritmo tiene la ventaja de no estar basado en un bucle comparado con el algoritmo de Cohen-Sutherland.

Ejemplo

Descripción

Queremos mostrar un segmento, pero únicamente dentro de nuestra vista representada por un triángulo descrito con los siguientes vértices: { (2,3), (3,-3), (-4,-2) }. Tenemos un segmento definido por la siguiente pareja de sus puntos extremos:
A = (-3,-1) y B = (1,1)

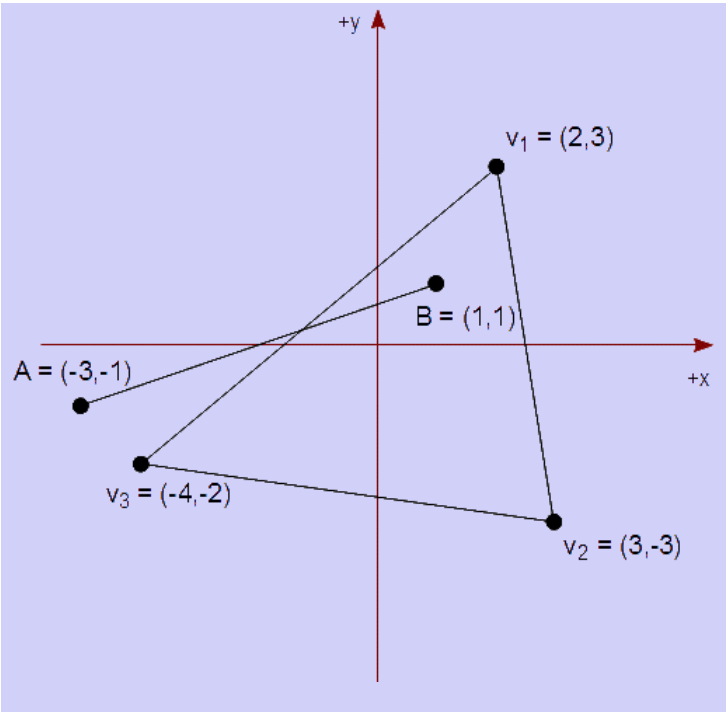


Figura 18 - Ejemplo del Algoritmo de Cyrus-Beck

Solución

Calculamos los vectores normales - sin normalizar - de todas las aristas de nuestro triángulo de recorte:

$\mathbf{v_{12}} = (\ 3, -3\ ) - (\ 2,\ 3\ ) = (\ \mathbf{1, -6}\ ) \Rightarrow \mathbf{N_1} = (\ \mathbf{6, 1}\ )$

$\mathbf{v_{23}} = (\ -4, -2\ ) - (\ 3, -3\ ) = (\ \mathbf{-7,\ 1}\ ) \Rightarrow \mathbf{N_2} = (\ \mathbf{-1, -7}\ )$

$\mathbf{v_{31}} = (\ 2,\ 3\ ) - (\ -4, -2\ ) = (\ \mathbf{6,\ 5}\ ) \Rightarrow \mathbf{N_3} = (\ \mathbf{-5, 6}\ )$

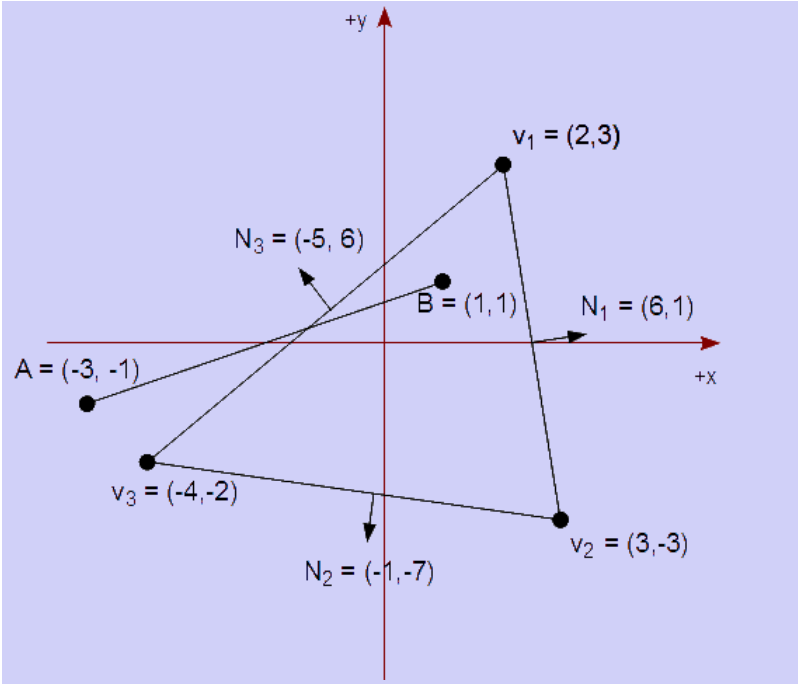


Figura 19 - Calculamos los vectores normales de cada arista

Calculamos el valor de *t* para la intersección de la línea que contiene el segmento AB con la línea generada por v_1v_2. Primero calculamos y comprobamos el denominador del cálculo de *t* por si es 0 y por tanto el segmento es paralelo a la arista:

$\mathbf{N_1 \cdot D} = (\ \mathbf{6,\ 1}\ ) \cdot (\ \mathbf{4,\ 2}\ ) = 6*4 + 1*2 = 26$

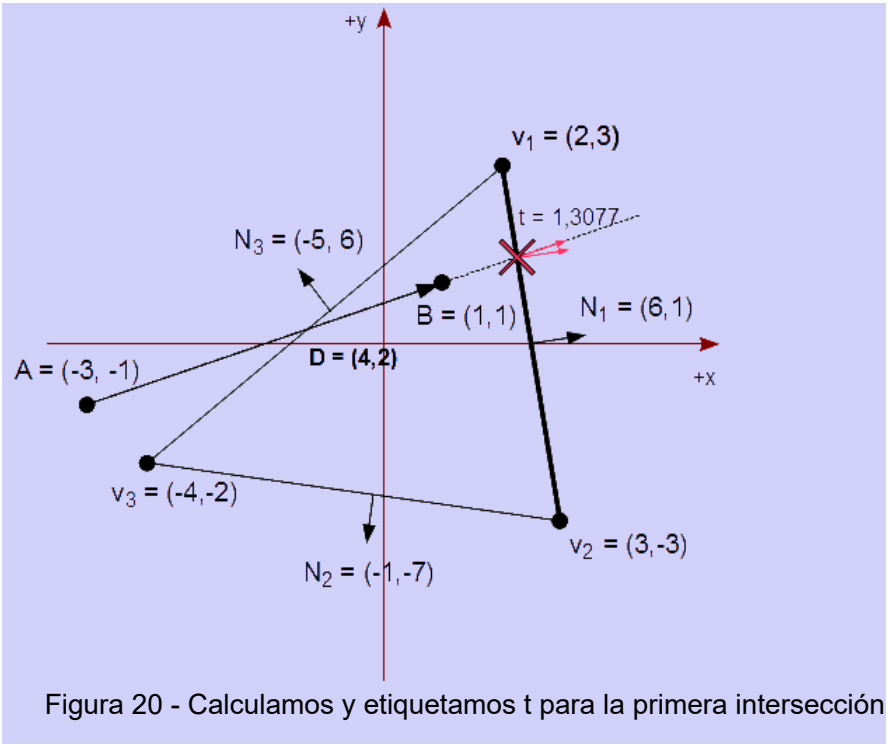


Figura 20 - Calculamos y etiquetamos t para la primera intersección

Como el denominador no es 0, calculamos el valor de *t* para t_s ya que el denominador es positivo por lo que se trata de un punto potencialmente saliente:

$$t = \frac{(6,\ 1) \cdot ((-3,\ -1) - (2,\ 3))}{-26}$$

$$= \frac{(6,\ 1) \cdot (-5,\ -4)}{-26}$$

$$= \frac{6*(-5) + 1*(-4)}{-26} = \frac{-34}{-26} = 1{,}3077$$

Como el valor de *t* no queda comprendido en [0,1], lo descartamos. El valor de t_s sigue siendo 1, en estos momentos, ya que queremos el menor valor para t_s. Esto es correcto, ya que 1 < 1,3077.

Calculamos el valor de *t* para la intersección de la línea que contiene el segmento AB con la línea generada por v_2v_3. Primero calculamos y comprobamos el denominador del cálculo de *t* por si es 0 y por tanto el segmento es paralelo a la arista:

$N_2 \cdot D = (-1,\ -7) \cdot (4,\ 2) = (-1)*4 + (-7)*2 = -18$

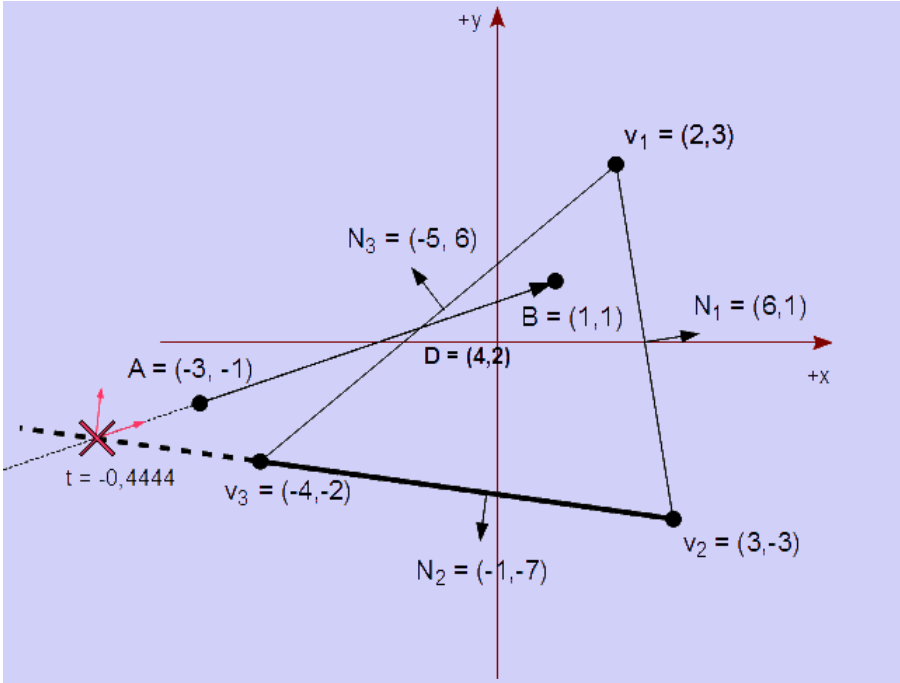


Figura 21 - Calculamos y etiquetamos t para la segunda intersección

Como el denominador no es 0, calculamos el valor de *t* para t_e ya que el denominador es negativo por lo que se trata de un punto potencialmente entrante:

$$t = \frac{(-1,\ -7) \cdot ((-3,\ -1) - (3,\ -3))}{-(-18)}$$

$$= \frac{(-1,\ -7) \cdot (-6,\ 2)}{18}$$

$$= \frac{(-1)*(-6) + (-7)*2}{18} = \frac{-8}{18} = -0{,}4444$$

Como el valor de *t* no queda comprendido en [0,1], lo descartamos. El valor de t_e sigue siendo 0, en estos momentos, ya que queremos el mayor valor para t_e. Esto es correcto, ya que -0,4444 < 0.

Calculamos el valor de *t* para la intersección de la línea que contiene el segmento AB con la línea generada por v_3v_1. Primero calculamos y comprobamos el denominador del cálculo de *t* por si es 0 y por tanto el segmento es paralelo a la arista:

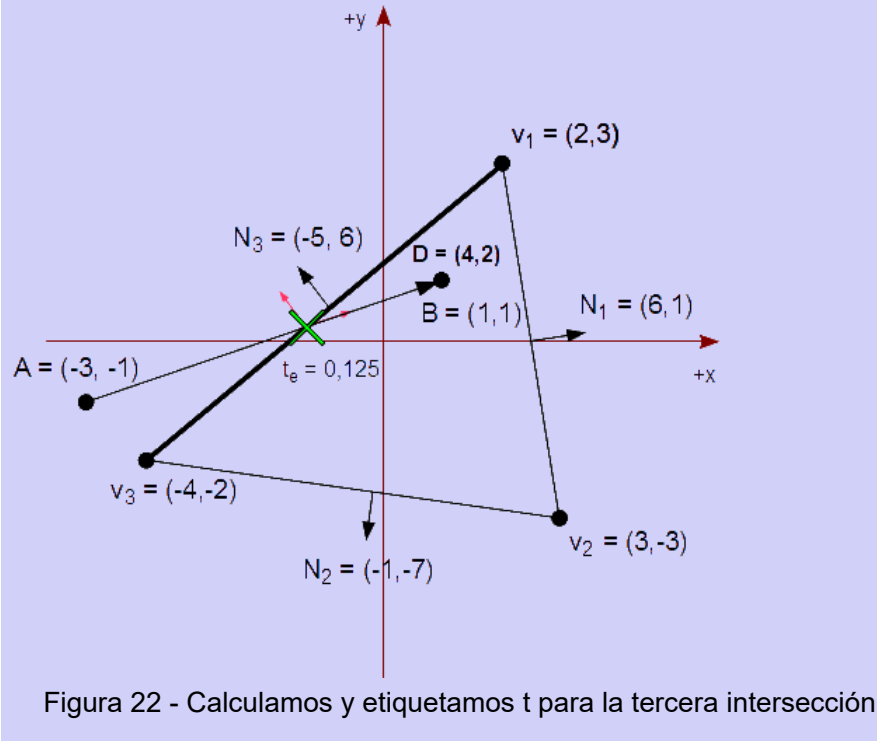


Figura 22 - Calculamos y etiquetamos t para la tercera intersección

$$N_3 \cdot D = (-5, 6) \cdot (4, 2) = (-5)*4 + 6*2 = -8$$

Como el denominador no es 0, calculamos el valor de *t* para t_e ya que el denominador es negativo por lo que se trata de un punto potencialmente entrante:

$$t = \frac{(-5, 6) \cdot ((-3, -1) - (-4, -2))}{-(-8)}$$

$$= \frac{(-5, 6) \cdot (1, 1)}{8}$$

$$= \frac{(-5)*1 + 6*1}{8} = \frac{1}{8} = 0{,}125$$

Como el valor de *t* queda comprendido en [0,1], actualizamos el valor de t_e para que sea 0,125, ya que 0,125 > 0, su valor anterior.

Al terminar de inspeccionar todas las aristas del triángulo de recorte, comprobamos los valores de t_e y t_s, ya que son las cotas inferior y superior, respectivamente, de un intervalo: $t_e \leq t_s$. En nuestro caso, obtenemos el siguiente intervalo válido: [0,125, 1].

Pasamos a calcular los nuevos puntos extremos para el segmento recortado:

$$A' = (-3, -1) + ((-3, -1) - (1, 1)) * 0{,}125$$
$$= (-3, -1) + (4, 2) * 0{,}125$$
$$= (-3 + 0{,}5, -1 + 0{,}25)$$

```
   = ( -2,5, -0,75 )

B' = ( -3, -1 ) + (( -3, -1 ) - ( 1, 1 )) * 1
   = ( -3, -1 ) + ( 4, 2 ) * 1
   = ( -3 + 4, -1 + 2 )
   = (  1,  1 )
```

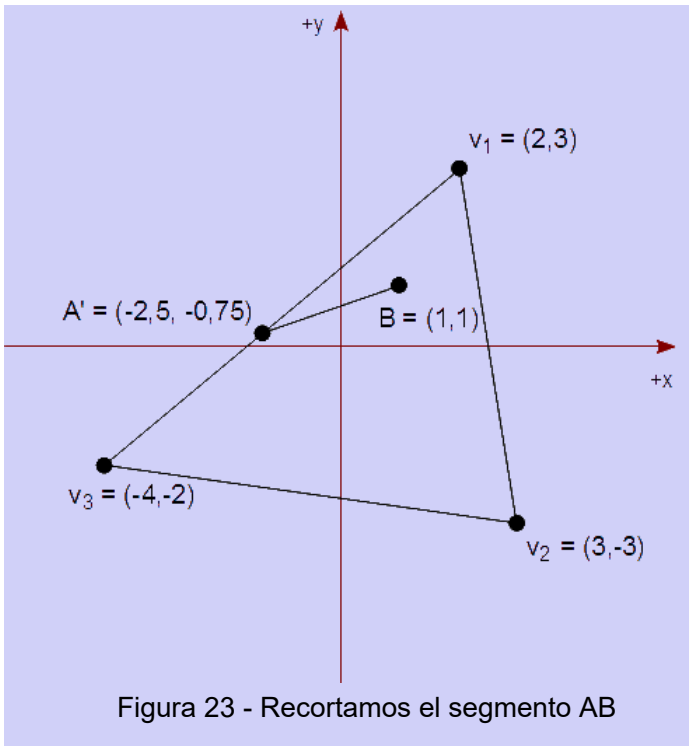


Figura 23 - Recortamos el segmento AB

Algoritmo

Tenemos varias funciones a destacar para implementar este algoritmo:

- La función principal es *Recortar()*, la cual aceptará tres parámetros: dos puntos extremos, P y Q, donde cada uno contiene las coordenadas (x,y) que representan el segmento a recortar, y un polígono convexo, R, que contiene una lista de vértices $\{v_1, v_2, v_3, ..., v_n\}$ en el sentido de las agujas del reloj para representar la región de recorte.
- La función *Recortar_Punto()* se invocará para tratar el segmento como un único punto. Básicamente, se realizará el algoritmo explicado anteriormente.
- La función *Calcular_Normal()* es necesaria si no tenemos una lista de normales de las aristas del polígono de recorte, R.
- Las funciones *mayor()* y *menor()* determinan cuáles de los dos parámetros dados son el mayor y el menor, respectivamente.
- La función *Calcular_Punto()* sirve para obtener un punto en una línea recta según la ecuación paramétrica. Esta función requiere dos puntos extremos de la línea y el parámetro, *t*.

Para *Recortar()*, el algoritmo es:

```
booleano Recortar( ref Punto P, ref Punto Q, PolígonoConvexo R )
        1. Si P = Q, entonces
                2. resultado ← Recortar_Punto( P, R )
                3. Terminar( resultado )
        4. Si no, entonces
                5. te ← 0
                6. ts ← 1
                7. Para cada vértice, vi, de R, hacer
                        8. N ← Calcular_Normal( vi+1 - vi )
                        9. numerador ← N·(P - vi)
```

```
                        10. denominador ← -N·(Q - P)
                        11. Si denominador ≠ 0, entonces
                                12. t ← numerador / denominador
                                13. Si denominador < 0, entonces
                                        14. te ← mayor( te, t )
                                15. Si no, entonces
                                        16. ts ← menor( ts, t )
                        17. Si no, compruebe que numerador > 0,
                                18. Terminar( falso )
                19. Si te > ts, entonces
                        20.Terminar( falso )
                21. Si no, entonces
                        22. Pe ← Calcular_Punto( P, Q, te )
                        23. Ps ← Calcular_Punto( P, Q, ts )
                        24. P ← Pe
                        25. Q ← Ps
        26. Terminar( verdadero )
```

Si el segmento es aceptado, entonces el algoritmo terminará con el valor booleano *verdadero*. Además, los extremos, P y Q, contendrán los valores recortados por este mismo algoritmo. Si el segmento es rechazado, el algoritmo termina con el valor *falso*, lo cual implica que los parámetros, P y Q, no son alterados. En cualquier caso, se pasan estos dos parámetros por referencia, para así poder modificarlos.

A continuación presentamos el algoritmo para *Calcular_Normal()*, el cual requiere un vector, **v**:

```
Vector Calcular_Normal( Vector v )
        1. vector N ← ( -v.y, v.x )
        2. N ← Normalizar( N )
        3. Terminar( N )
```

Este cálculo nos sirve si el vector dado sigue el convenio de ir en el sentido de las agujas del reloj. La función *Normalizar()* se refiere a la operación vectorial para convertir un vector en unitario. Sin embargo, en este algoritmo, no es necesario normalizar estos vectores normales, por lo que podemos conseguir una optimización para el algoritmo de Cyrus-Beck.

Por último, tenemos el algoritmo para *Calcular_Punto()*, que requiere dos puntos, P y Q, y un valor para *t*:

```
Punto Calcular_Punto( Punto P, Punto Q, real t )
        1. Punto A ← P + (Q-P) t
        2. Terminar( A )
```

Algoritmo de Liang-Barsky

Este algoritmo se basa en el algoritmo propuesto por Cyrus-Beck. La diferencia es que el algoritmo de Liang-Barsky aplica ciertas interpretaciones geométricas y simplificaciones al basarse en un rectángulo vertical de recorte. Con un rectángulo vertical, podemos definir los vectores normales sin problemas al igual que los vectores en las aristas. Veamos una tabla de ciertos valores y las simplificaciones que podemos realizar.

Arista de Recorte	**Normal, N**	**Vértice, V**	**P_0-V**	**$D = P_1-P_0$**	**$t = -N\cdot(P_0 - V) / N\cdot D$**
izquierda: $x = x_{izq}$	(-1,0)	(x_{izq}, V_y)	(x_0-x_{izq}, y_0-V_y)	(x_1-x_0, y_1-y_0)	$-(x_0-x_{izq}) / (x_1-x_0)$
derecha: $x = x_{der}$	(1,0)	$(x_{der}, V_i.y)$	(x_0-x_{der}, y_0-V_y)	(x_1-x_0, y_1-y_0)	$(x_0-x_{der}) / -(x_1-x_0)$

superior: $y = y_{sup}$	(0,-1)	$(V_i.x, y_{sup})\ (x_0-V_x, y_0-y_{sup})\ (x_1-x_0, y_1-y_0)\ -(y_0-y_{sup}) / (y_1-y_0)$
inferior: $y = y_{inf}$	(0,1)	$(V_i.x, y_{inf})\ (x_0-V_x, y_0-y_{inf})\ (x_1-x_0, y_1-y_0)\ (y_0-y_{inf}) / -(y_1-y_0)$

Según esta tabla, vemos que el cálculo de *t* se reduce a una división de distancias entre coordenadas homogéneas: anchuras para el recorte de las coordenadas en X y alturas para el recorte de las coordenadas en Y. El numerador pasa a ser la distancia del punto extremo del segmeneto a la arista. Esta distancia equivale a la misma cantidad que usamos en el método de Cohen-Sutherland al calcular el código binario para cada punto extremo. También podemos ver que el denominador es la anchura, *dx*, o la altura, *dy*. Según el signo de estas longitudes, el segmento es PE o PS, y por ello los signos del numerador y del denominador se deben conservar, para que el algoritmo funcione correctamente.

Ejemplo

Descripción

Retomamos el mismo ejemplo del método exhaustivo. Tenemos un rectángulo de recorte cuya diagonal es de (-1,3) a (3,-3) que representa nuestra vista. Queremos mostrar el segmento AB, en tal rectángulo de recorte, descrito por los puntos,

$A = (-2, 1)$ y $B = (2, 2)$

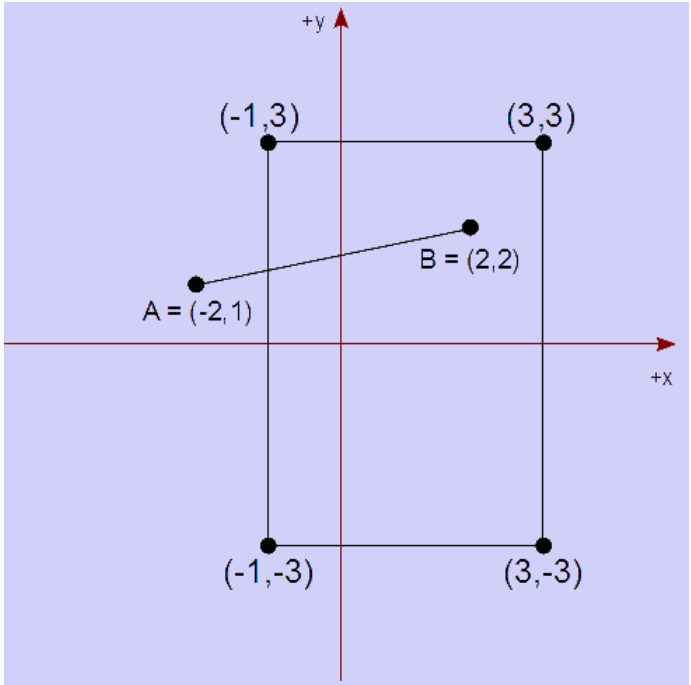


Figura 24 - Ejemplo del Algoritmo de Liang-Barsky

Solución

Calculamos el vector **D** de A a B:

$$\mathbf{D} = B - A = (\ 2,\ 2\ ) - (\ -2,\ 1\ ) = \mathbf{(\ 4,\ 1\ )}$$

Comprobamos el caso trivial en el que los dos puntos representan un mismo punto visible, en lugar de un segmento. Como **D ≠ (0,0)**, tenemos un segmento.

Calculamos la intersección del segmento AB con la arista izquierda. Como el denominador, $D_x = 4$, es positivo, calculamos *t*:

$$t = \frac{R.x_{izq} - A_x}{-D_x} = \frac{-1 - (-2)}{4} = \frac{1}{4} = 0,25$$

Como $t > t_e$, actualizamos $t_e = 0,25$. El intervalo de *t* por ahora es [0,25, 1].

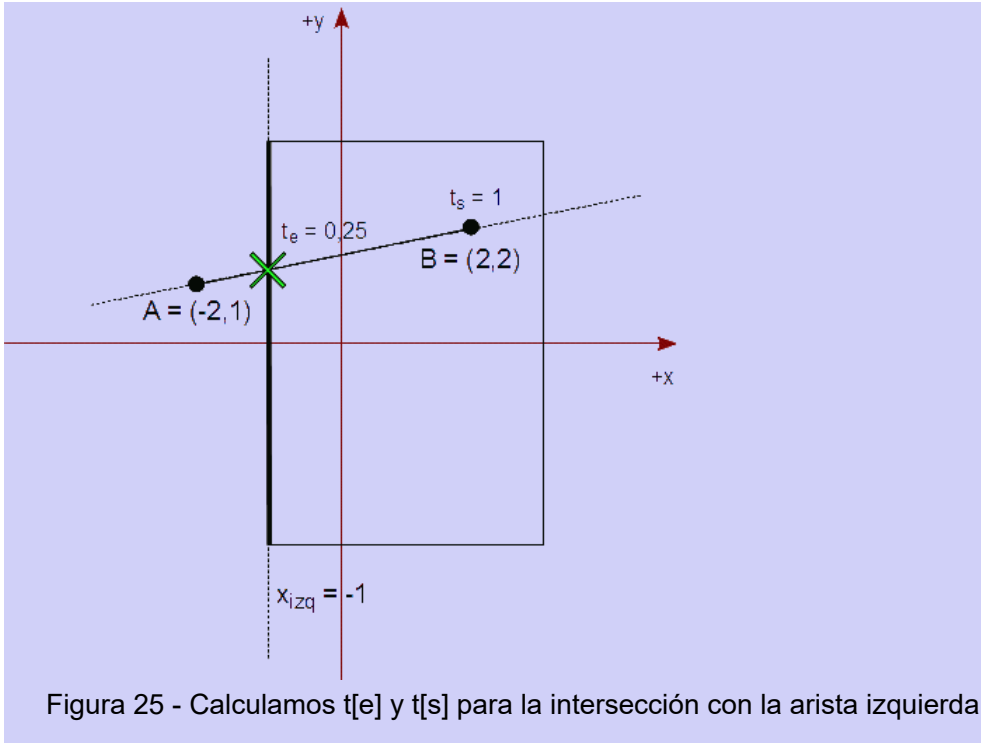


Figura 25 - Calculamos t[e] y t[s] para la intersección con la arista izquierda

Calculamos la intersección del segmento AB con la arista derecha. Usamos el denominador, $-D_x = -4$, que al ser negativo, calculamos *t*:

$$t = \frac{A_x - R.x_{der}}{-D_x} = \frac{-2 - 3}{-4} = \frac{-5}{-4} = 1,25$$

Como $t > t_s$, no actualizamos t_s. El intervalo de *t* por ahora sigue siendo [0,25, 1].

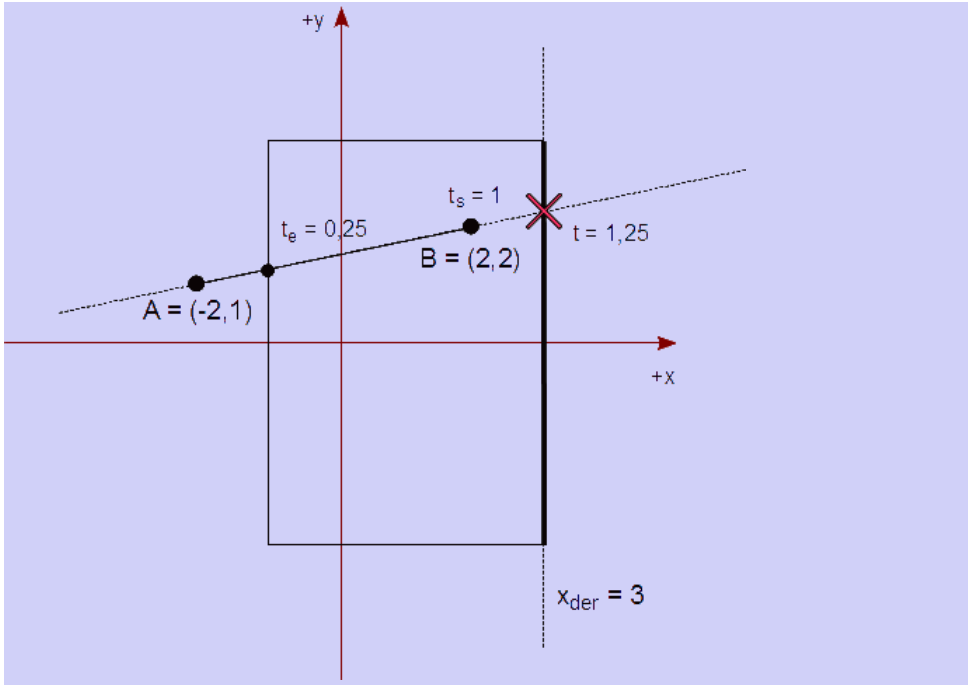


Figura 26 - Calculamos t[e] y t[s] para la intersección con la arista derecha

Calculamos la intersección del segmento AB con la arista inferior. Usamos el denominador, $D_y =$

1, que al ser positivo, calculamos *t*:

$$t = \frac{R.y_{inf} - A_y}{D_y} = \frac{-3 - 1}{1} = -4$$

Como $t < t_e$, no actualizamos t_e. El intervalo de *t* por ahora sigue siendo [0,25, 1].

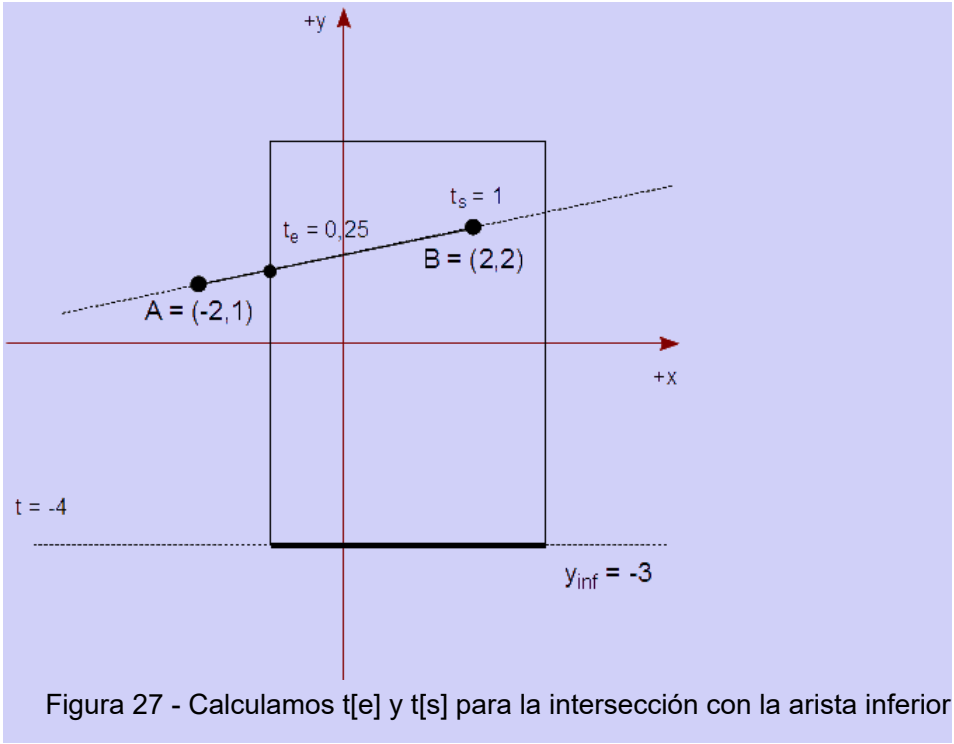


Figura 27 - Calculamos t[e] y t[s] para la intersección con la arista inferior

Calculamos la intersección del segmento AB con la arista superior. Usamos el denominador, $-D_y = -1$, que al ser negativo, calculamos *t*:

$$t = \frac{A_y - R.y_{sup}}{-D_y} = \frac{1 - 3}{-1} = 2$$

Como $t > t_s$, no actualizamos t_s. El intervalo de *t* por ahora sigue siendo [0,25, 1].

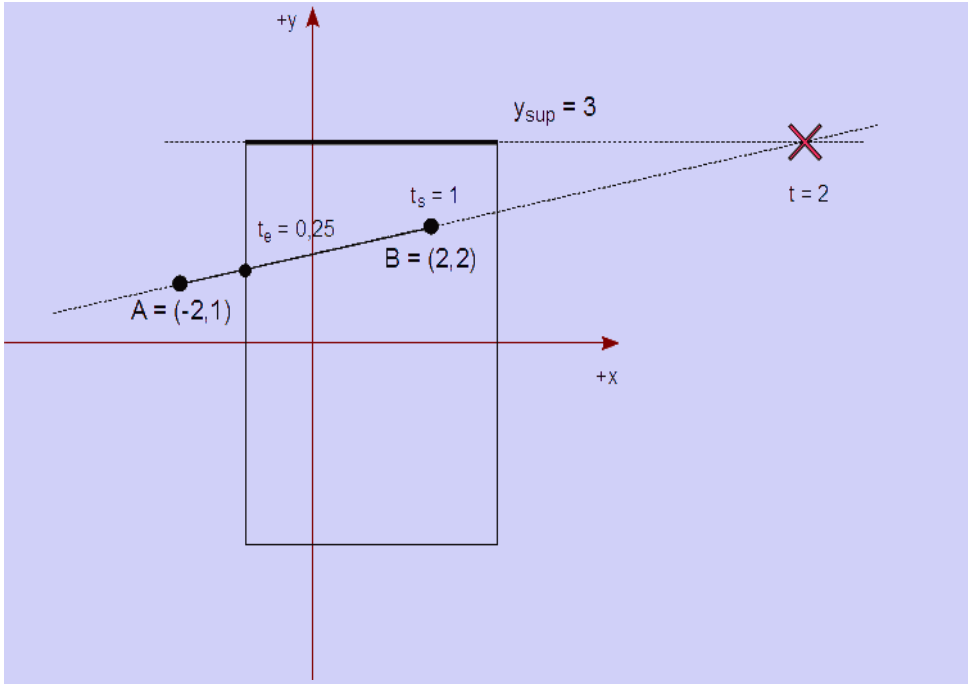


Figura 28 - Calculamos t[e] y t[s] para la intersección con la arista superior

Como t_s no fue actualizado, no necesitamos que realizar ningún cáculo. Como t_e fue actualizado, calculamos el punto entrante de intersección, A:

```
A' = A + te * ( Dx, Dy )
   = ( -3,  1 ) + 0,25 * ( 4,  1 )
   = ( -2,  1,25 )
```

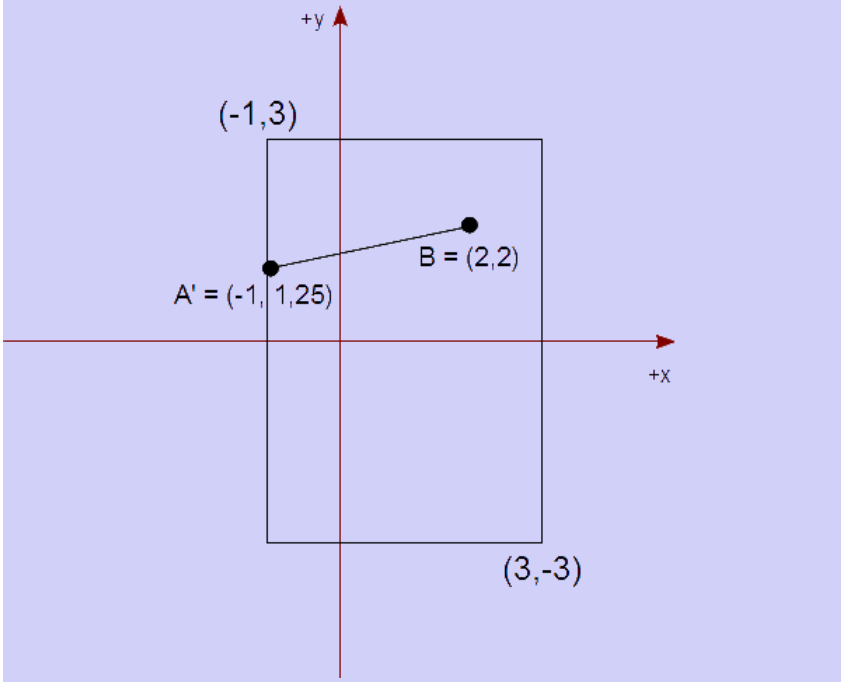


Figura 29 - Calculamos los puntos extremos a partir de t[e] y t[s]

Algoritmo

Tenemos varias funciones a destacar para implementar este algoritmo:

- La función principal es *Recortar()*, la cual aceptará tres parámetros: dos puntos extremos, P y Q, donde cada uno contiene las coordenadas (x,y) que representan el segmento a recortar, y un rectángulo vertical, R, que contiene los elementos $\{x_{izq},y_{sup}, x_{der},y_{inf}\}$ representando las coordenadas de las dos esquinas izquierda superior y derecha inferior.
- La función *Recortar_Punto()* se invocará para tratar el segmento como un único punto. Básicamente, se realizará el algoritmo explicado anteriormente. El resultado es un valor booleano que indicará si el punto fue recortado (verdadero) o no (falso).
- La función *Calcular_Parámetros()* servirá para calcular los parámetros, t_e y t_s. Necesitamos pasar el numerador, el denominador, y los parámetros, t_e y t_s, que serán modificados. Esta función nos indicará si estos parámetros, t_e y t_s, han sido modificados y por tanto, el segmento es aceptado (verdadero) o rechazado (falso).

Para *Recortar()*, el algoritmo es:

```
booleano Recortar( ref Punto P, ref Punto Q, Rectángulo R )
        1. dx ← Q.x - P.x
        2. dy ← Q.y - P.y
        3. resultado ← falso
        4. Si dx = 0,  y  Si dy = 0,  y  Si Recortar_Punto( P, R ) = verdadero, entonces
                5. Q ← P
                6. resultado ← verdadero
        7. Si no, entonces
                8. te ← 0
                9. ts ← 1
                10. Si Calcular_Parámetros( R.xizq-P.x, dx, te, ts ) = verdadero,
```

```
entonces
                        11. Si Calcular_Parámetros( P.x-R.x_der, -dx, t_e, t_s ) =
verdadero, entonces
                                12. Si Calcular_Parámetros( R.y_inf-P.y, dy, t_e, t_s ) =
verdadero, entonces
                                        13. Si Calcular_Parámetros( P.y-R.y_sup, -dy, t_e,
t_s ) = verdadero, entonces
                                                14. resultado ← verdadero
                                                15. Si t_s < 1, entonces
                                                        16. Q.x ← P.x + t_s dx
                                                        17. Q.y ← P.y + t_s dy
                                                18. Si t_e > 0, entonces
                                                        19. P.x ← P.x + t_e dx
                                                        20. P.y ← P.y + t_e dy
        21. Terminar( resultado )
```

Si el segmento es aceptado, entonces el algoritmo terminará con el valor booleano *verdadero*. Además, los extremos, P y Q, contendrán los valores recortados por este mismo algoritmo. Si el segmento es rechazado, el algoritmo termina con el valor *falso*, lo cual implica que los parámetros, P y Q, no son alterados. En cualquier caso, se pasan estos dos parámetros por referencia, para así poder modificarlos.

A continuación presentamos el algoritmo para *Calcular_Parámetros()*, el cual requiere el numerador, el denominador, t_e, y t_s, los cuales estos dos últimos se pasan por referencia:

```
booleano Calcular_Parámetros( real numerador, real denominador, ref real t_e, ref real t_s
)
        1. Si denominador > 0, entonces
                2. t ← numerador / denominador
                3. Si t > t_s
                        4. Terminar( falso )
                5. Si no, compruebe que t > t_e
                        6. t_e ← t
        7. Si no, compruebe que denominador < 0,
                8. t ← numerador / denominador
                9. Si t < t_e
                        10. Terminar( falso )
                11. Si no, compruebe que t < t_s
                        12. t_s ← t
        13. Si no, compruebe que numerador > 0,
                14. Terminar( falso )
        15. Terminar( verdadero )
```

Algoritmo de Nicholl-Lee-Nicholl

Este algoritmo intenta corregir los problemas de los algoritmos anteriores para recortar segmentos en un rectángulo vertical. El algoritmo de Cohen-Sutherland se basa en realizar varias comprobaciones, pero acaba calculando varias intersecciones que pueden o no servir. Los algoritmos de Cyrus-Beck y Liang-Barsky se basan en calcular varias intersecciones y luego comprobar cuáles sirven. El algoritmo de Nicholl-Lee-Nicholl elimina la necesidad de realizar todos estos cálculos de intersecciones favoreciendo las comprobaciones para dar con la intersección correcta, si existe. Al final, este algoritmo es más rápido que los mencionados anteriormente.

Para recortar un segmento con los extremos P y Q, necesitamos determinar las intersecciones, si existen, con las aristas. Comenzamos dividiendo la geometría en nueve regiones al extender horizontal y verticalmente las aristas del rectángulo vertical, como hicimos en el algoritmo de

Cohen-Sutherland. Sin embargo, sólo necesitamos considerar tres casos de la posición del punto extremo, P, porque los demás casos son simétricos a uno de estos tres casos principales. Estos casos son:

1. P es interior al rectángulo de recorte,

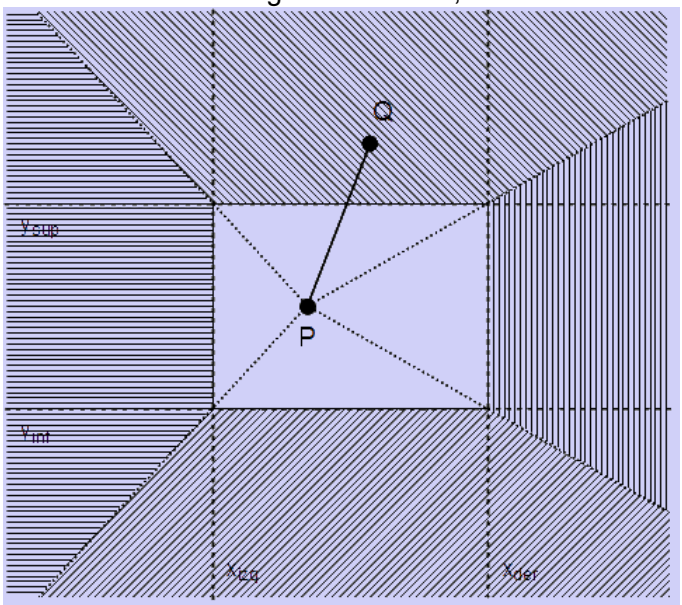


Figura 30 - Se divide el plano en varias regiones con el punto P interior

2. P está en una región exterior que hace esquina con el rectángulo de recorte, o

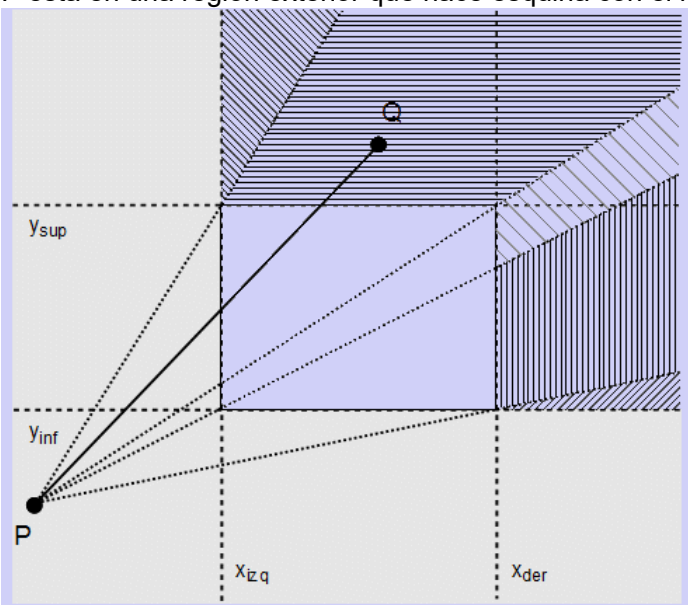


Figura 31 - Se divide el plano en varias regiones con el punto P en una esquina

3. P está en una región exterior que toca una arista del rectángulo de recorte.

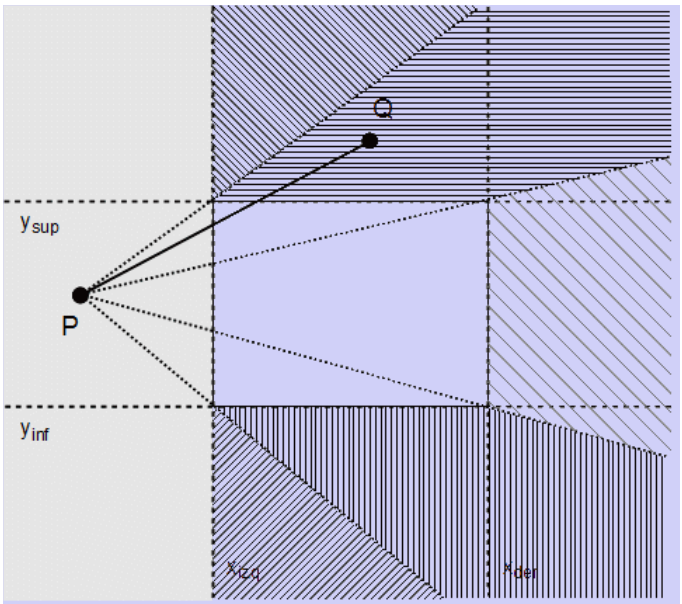


Figura 31 - Se divide el plano en varias regiones con el punto P en un lateral

En cada caso, podemos dividir nuevamente nuestro espacio en diferentes regiones para determinar dónde se encuentra el otro punto extremo, Q. Las nuevas divisiones se basan en crear vectores desde el punto conocido, P, a cada uno de los vértices del rectángulo de recorte. Al determinar si el punto extremo, Q, está a la izquierda de uno de estos vectores, de P a un vértice, podemos reducir las posibles regiones hasta encontrar la que contiene el punto, Q. Una vez que hayamos encontrado el punto, Q, y la región en la que se encuentra, podemos aplicar la ecuación de la intersección precisa para el caso en cuestión. Para los dos últimos casos, podemos pensar que los vectores de P a los vértices del rectángulo de recorte que son exteriores forman un "campo de visión". Cualquier punto dentro de este campo será visible y por tanto el segmento cruzará el rectángulo. Para el primer caso, usamos estos vectores simplemente para dividir el espacio en regiones para encontrar rápidamente el otro punto, Q, ya que sabemos de antemano que el segmento será visible, parcial o completamente.

Ejemplo

Descripción

Retomamos el mismo ejemplo del método exhaustivo. Tenemos un rectángulo de recorte cuya diagonal es de (-1,3) a (3,-3) que representa nuestra vista. Queremos mostrar el segmento AB, en tal rectángulo de recorte, descrito por los puntos,

```
A = ( -2, 1 )  y  B = ( 2, 2 )
```

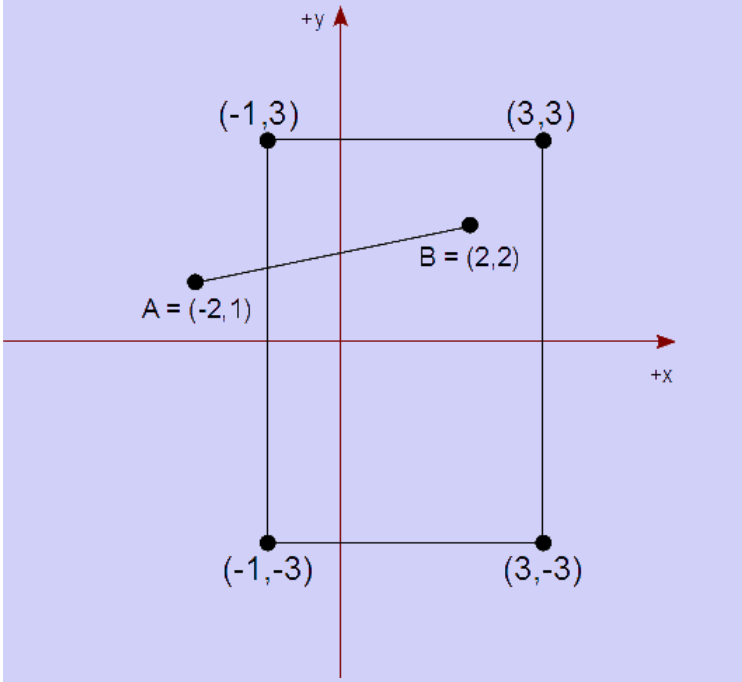


Figura 33 - Ejemplo del Algoritmo de Nicholl-Lee-Nicholl

Solución

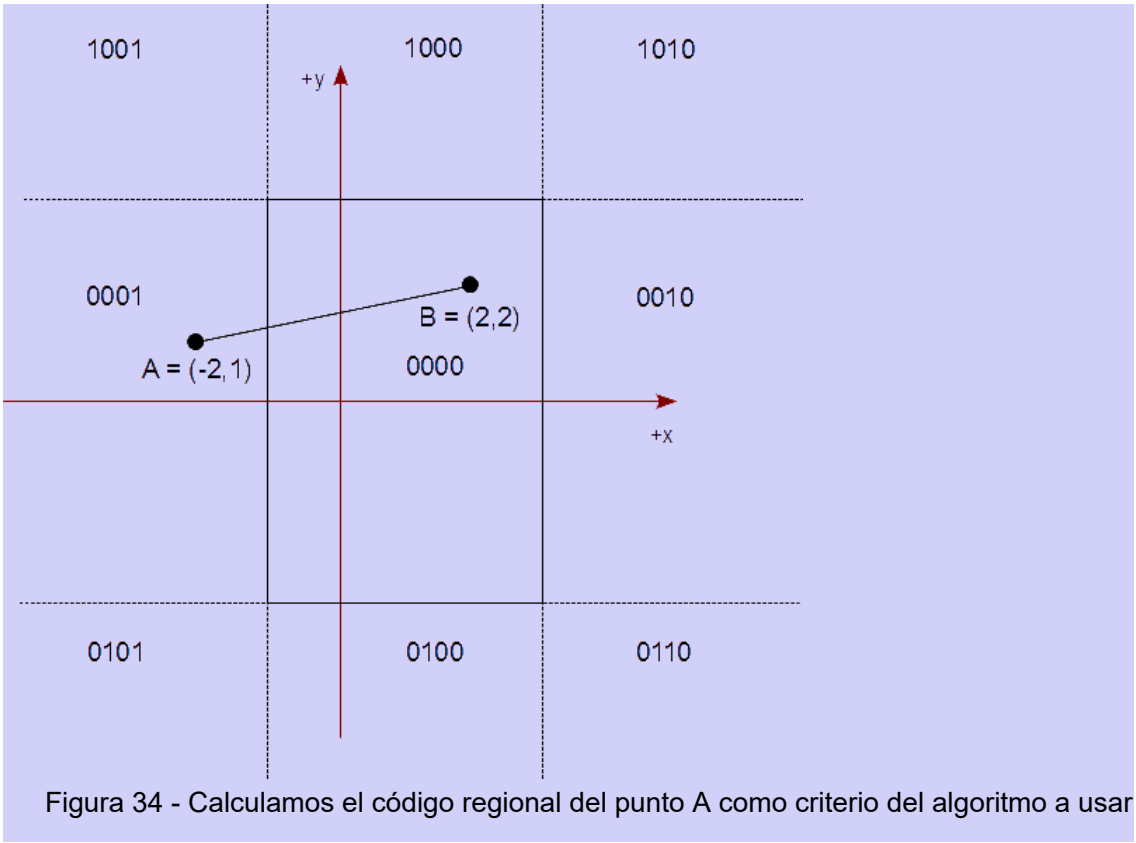


Figura 34 - Calculamos el código regional del punto A como criterio del algoritmo a usar

Calculamos el código regional del punto A como hicimos en el método de Cohen-Sutherland:

A : 0001

Vemos que A está en una región lateral del rectángulo vertical de recorte. Esto significa que estamos ante el caso #2, en la figura 32.

Ahora nos toca descubrir la ubicación del punto B para determinar si hace falta recortar el segmento o no. Si es necesario recortar el segmento, debemos descubrir exactamente el punto de intersección. En principio, podemos comprobar si B está en alguna región vertical a la de A. Esto es,

$B_x < R.x_{izq} \Rightarrow 2 < -1$

Como no es cierto, debemos continuar con el algoritmo.

Averiguamos si B está a la derecha o a la izquierda de una línea que contiene A y la esquina izquierda superior del rectángulo de recorte. Para ello, calculamos los siguientes vectores:

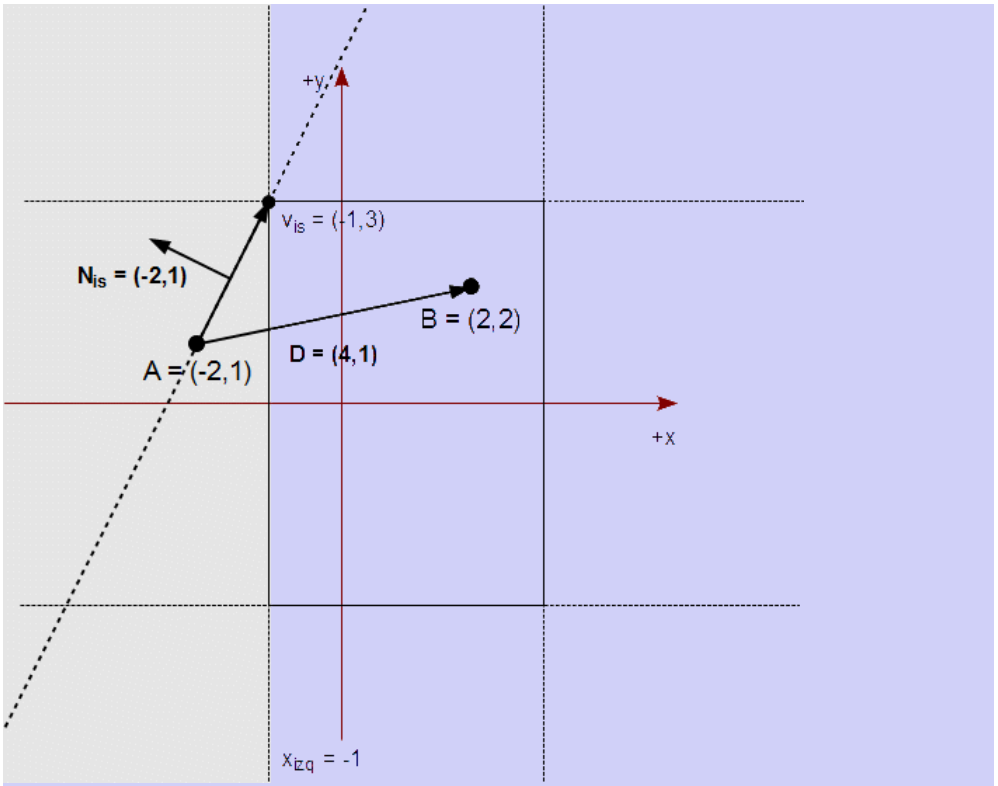


Figura 35 - Determinamos si B está a la izquierda o derecha de la línea AVis

D = B - A = (2, 2) - (-2, 1) = **(4, 1)**

$\mathbf{v_{is}}$ = ($R.x_{izq}$, $R.y_{sup}$) - A = (-1, 3) - (-2, 1) = **(1, 2)**

Aplicamos el mismo método que usamos en el algoritmo de Cyrus-Beck para determinar si un punto estaba a un lado o a otro de una línea recta. Calculamos el vector normal de $\mathbf{v_{is}}$ - sin normalizar:

$\mathbf{N_{is}}$ = **(-2, 1)**

El signo de **N** · **D** nos indicará si B está a la izquierda o a la derecha:

(-2, 1) · **(4, 1)** = (-2)*4 + 1*1 = -7 < 0

Como el signo es negativo, B está a la derecha y por tanto debemos continuar con el algoritmo.

Averiguamos si B está a la

derecha o a la izquierda de una línea que contiene A y la esquina derecha superior del rectángulo de recorte. Para ello, calculamos el vector de tal línea:

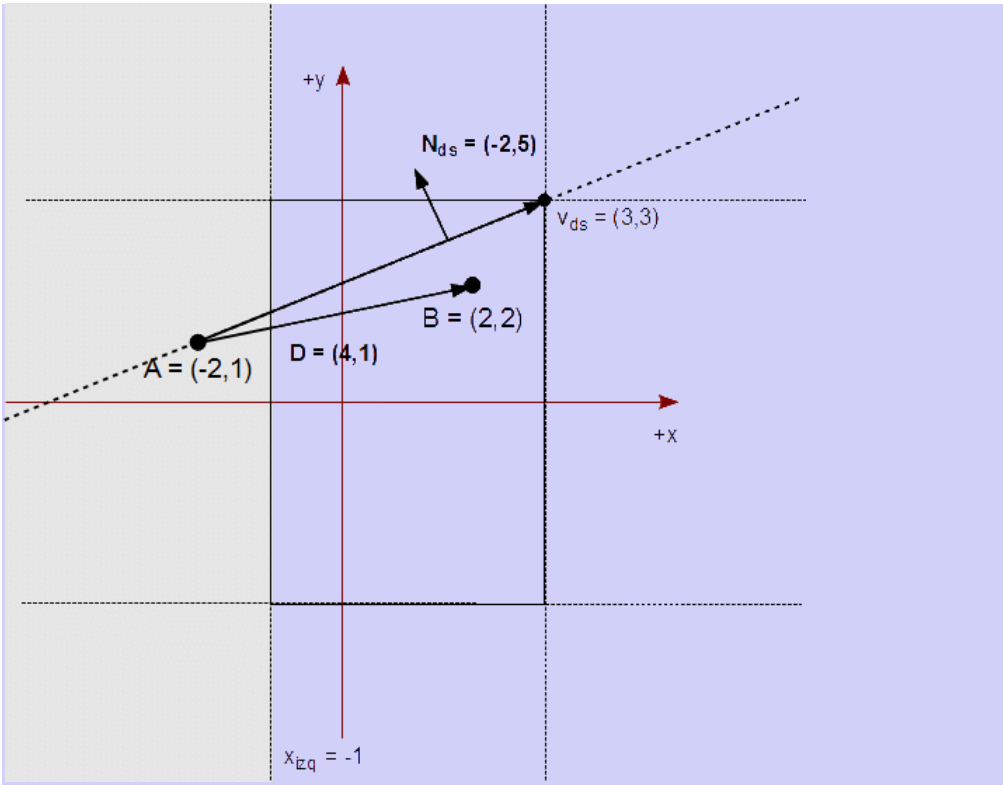


Figura 36 - Determinamos si B está a la izquierda o derecha de la línea AVds

$\mathbf{v_{ds}} = (R.x_{der}, R.y_{sup}) - A = (3, 3) - (-2, 1) = \mathbf{(5, 2)}$

Calculamos el vector normal de $\mathbf{v_{ds}}$:

$\mathbf{N_{ds}} = \mathbf{(-2, 5)}$

El signo de **N** · **D** nos indicará si B está a la izquierda o a la derecha:

$\mathbf{(-2, 5)} \cdot \mathbf{(4, 1)} = (-2)*4 + 5*1 = -3 < 0$

Como el signo es negativo, B está a la derecha y por tanto debemos continuar con el algoritmo.

Averiguamos si B está a la derecha o a la izquierda de una línea que contiene A y la esquina derecha inferior del rectángulo de recorte. Para ello, calculamos el vector de tal línea:

$\mathbf{v_{di}} = (R.x_{der}, R.y_{inf}) - A = (3, -3) - ($

$-2, 1) = (5, -4)$

Calculamos el vector normal de $\mathbf{v_{di}}$:

$\mathbf{N_{di}} = (4, 5)$

El signo de $\mathbf{N} \cdot \mathbf{D}$ nos indicará si B está a la izquierda o a la derecha:

$(4, 5) \cdot (4, 1) = 4*4 + 5*1 = 21 > 0$

Como el signo es positivo, B está a la izquierda de esta línea.

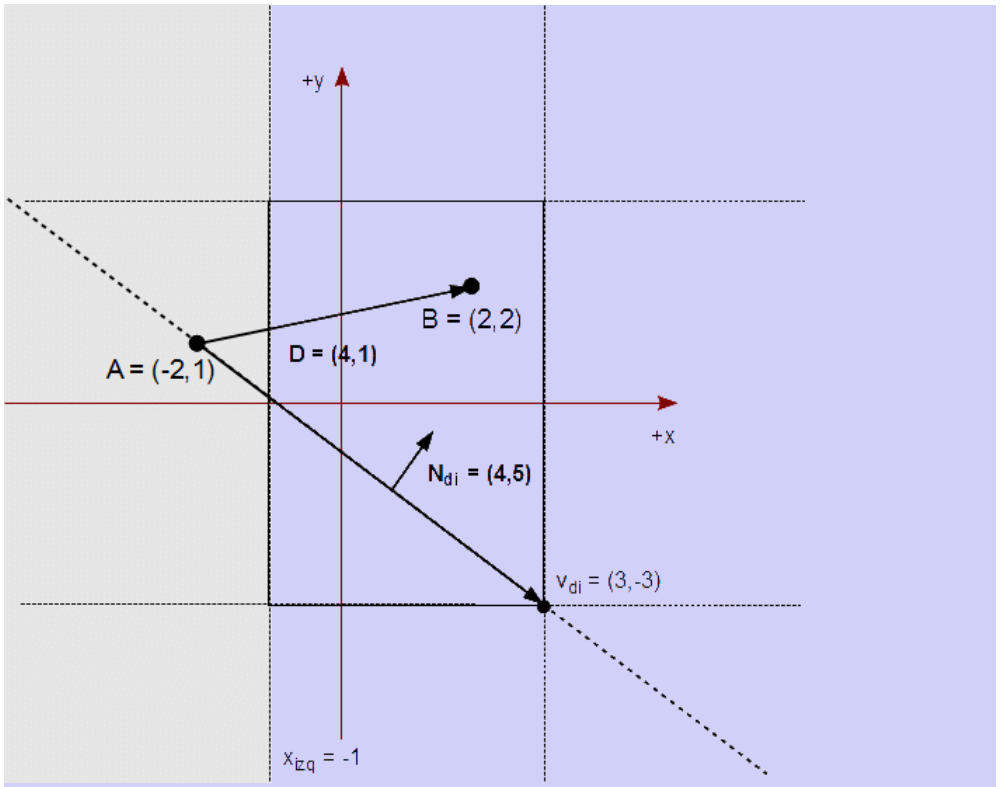


Figura 37 - Determinamos si B está a la izquierda o derecha de la línea AVdi

Como B está a la izquierda de $\mathbf{v_{di}}$ pero a la derecha de $\mathbf{v_{ds}}$, entonces sabemos que debemos recortar A con la arista izquierda:

$A'_x = R.x_{izq} \Rightarrow A'_x = -1$

$A'_y = A_y - (A_x - R.x_{izq}) * D_y / D_x$

$= 1 - (-2 - (-1))$

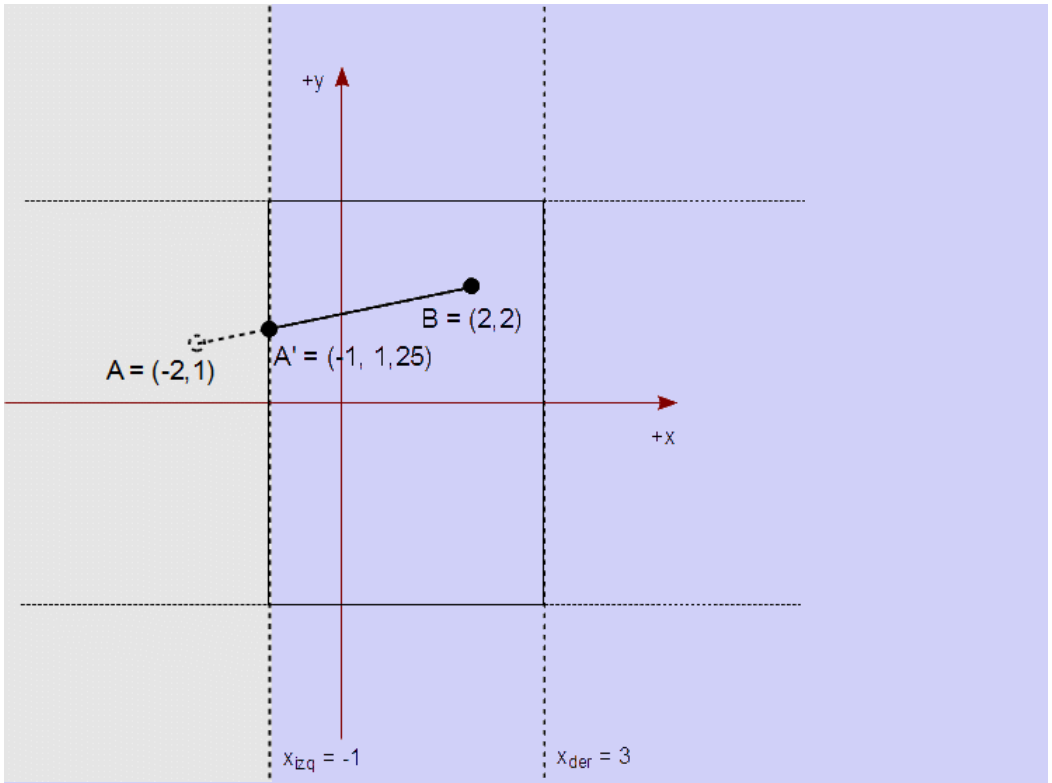


Figura 38 - Determinamos si B está a la izquierda o derecha de la arista derecha

$* 1 / 4 \Rightarrow A'_y = 1,25$

B está o bien dentro del rectángulo de recorte o bien fuera. Esto implica que B está a la izquierda de la arista derecha del rectángulo, y por tanto es interior, o a la derecha, y por tanto se debe recortar. Realizamos una comprobación sencilla:

$B_x < R.x_{der} \Rightarrow 2 < 3$

Por lo tanto, B está dentro y no necesitamos realizar ninguna intersección. El segmento AB se recorta quedando como,

$A'B = (-1, 1,25)$

Algoritmo

La función principal es *Recortar()*, la cual aceptará tres parámetros: dos puntos extremos, P y Q, donde cada uno contiene las coordenadas (x,y) que representan el segmento a recortar, y un rectángulo vertical, R, que contiene los elementos $\{x_{izq},y_{sup}, x_{der},y_{inf}\}$ representando las coordenadas de las dos esquinas izquierda superior y derecha inferior. Para hallar la región en la que se encuentra el punto extremo, P, del segmento a recortar, podemos usar el algoritmo de *Calcular_Código()* de Cohen-Sutherland para conseguir un código representativo de la región. Por ello, volvemos a definir los códigos binarios para: **SUPERIOR**=8, **INFERIOR**=4, **DERECHA**=2, e **IZQUIERDA**=1.

Como existen tres casos principales, definimos tres algoritmos especializados para recortar un segmento: *Recortar_Interior()*, *Recortar_Izquierda_Inferior()*, y *Recortar_Izquierda()*. Para los otros seis casos, modificamos los puntos extremos, P y Q, aplicando traslaciones y reflejo debido a las propiedades simétricas para convertir cada caso en uno principal. Cada algoritmo especializado hará uso de otros dos tipos de algoritmos:

- La función *Izquierda()* nos indicará si un punto está a la izquierda (*verdadero*) de una línea representada por dos puntos o no (*falso*). La función *Derecha()* realizará una comprobación parecida si un punto está a la derecha (*verdadero*) de una línea o no (*falso*).
- Las funciones *Intersección_Vertical()* e *Intersección_Horizontal()* calcularán la intersección entre el segmento, representado como un vector de A a B, y una arista vertical u horizontal del rectángulo de recorte, respectivamente.

```
booleano Recortar( ref Punto P, ref Punto Q, Rectángulo R )
        1. tipo ← Calcular_Código( P, R )
        2. Si no, compruebe que tipo = INTERIOR,      // Caso #1
                3. aceptar ← Recortar_Interior( P, Q, R )
        4. Si tipo = IZQUIERDA OR INFERIOR, entonces  // Caso #2
                5. aceptar ← Recortar_Izquierda_Inferior( P, Q, R )
        6. Si no, compruebe que tipo = DERECHA OR INFERIOR,
                7. Tx ← (R.xizq + R.xder) / 2
                8. P.x ← Tx - P.x
                9. Q.x ← Tx - Q.x
                10. aceptar ← Recortar_Izquierda_Inferior( P, Q, R )
                11. P.x ← Tx - P.x
                12. Q.x ← Tx - Q.x
        13.Si no, compruebe que tipo = IZQUIERDA OR SUPERIOR,
                14. Ty ← (R.ysup + R.yinf) / 2
                15. P.y ← Ty - P.y
```

```
                16. Q.y ← T_y - Q.y
                17. aceptar ← Recortar_Izquierda_Inferior( P, Q, R )
                18. P.y ← T_y - P.y
                19. Q.y ← T_y - Q.y
        20. Si no, compruebe que tipo = DERECHA OR SUPERIOR,
                21. T_x ← (R.x_izq + R.x_der) / 2
                22. T_y ← (R.y_sup + R.y_inf) / 2
                23. P ← T - P
                24. Q ← T - Q
                25. aceptar ← Recortar_Izquierda_Inferior( P, Q, R )
                26. P ← T - P
                27. Q ← T - Q
        28. Si no, compruebe que tipo = IZQUIERDA,     // Caso #3
                29. aceptar ← Recortar_Izquierda( P, Q, R )
        30. Si no, compruebe que tipo = DERECHA,
                31. T_x ← (R.x_izq + R.x_der) / 2
                32. P.x ← T_x - P.x
                33. Q.x ← T_x - Q.x
                34. aceptar ← Recortar_Izquierda( P, Q, R )
                35. P.x ← T_x - P.x
                36. Q.x ← T_x - Q.x
        37. Si no, compruebe que tipo = INFERIOR,
                38. T_x ← R.x_izq
                39. T_y ← R.y_inf
                40. P ← T - P
                41. Q ← T - Q
                42. aceptar ← Recortar_Izquierda( P, Q, R )
                43. P ← T - P
                44. Q ← T - Q
        45. Si no, compruebe que tipo = SUPERIOR,
                46. T_x ← R.x_izq
                47. T_y ← R.y_sup
                48. P ← T - P
                49. Q ← T - Q
                50. aceptar ← Recortar_Izquierda( P, Q, R )
                51. P ← T - P
                52. Q ← T - Q
        53. Terminar( aceptar )
```

Usamos el operador **OR** para indicar la operación OR a nivel de bits.

Aquí presentamos los algoritmos particulares para el recorte según la región donde se encuentra P, empezando por *Recortar_Interior()*:

```
booleano Recortar_Interior( ref Punto P, ref Punto Q, Rectángulo R )
        1. aceptar ← verdadero
        2. PQ ← Q - P
        3. PVis ← (R.x_izq,R.y_sup) - P
        4. Si Izquierda( PQ, PVis ), entonces
                5. PVii ← (R.x_izq,R.y_inf) - P
                6. Si Izquierda( PQ, PVii ), entonces
                        7. PVdi ← (R.x_der,R.y_inf) - P
                        8. Si Izquierda( PQ, PVdi ), entonces
                                9. Si Q.x > R.x_der, entonces
                                        10. Q ← Intersección_Vertical( P, Q, R.x_der )
                        11. Si no, entonces
                                12. Si Q.y < R.y_inf, entonces
                                        13. Q ← Intersección_Horizontal( P, Q, R.y_inf )
                14. Si no, entonces
                        15. Si Q.x < R.x_izq, entonces
                                16. Q ← Intersección_Vertical( P, Q, R.x_izq )
```

```
        17. Si no, entonces
                18. PVds ← (R.x_der,R.y_sup) - P
                19. Si Izquierda( PQ, PVds ), entonces
                        20. Si Q.y > R.y_sup, entonces
                                21. Q ← Intersección_Horizontal( P, Q, R.y_sup )
                22. Si no, entonces
                        23. PVdi ← (R.x_der,R.y_inf) - P
                        24. Si Izquierda( PQ, PVdi ), entonces
                                25. Si Q.x > R.x_der, entonces
                                        26. Q ← Intersección_Vertical( P, Q, R.x_der )
                        27. Si no, entonces
                                28. Si Q.y < R.y_inf, entonces
                                        29. Q ← Intersección_Horizontal( P, Q, R.y_inf )
        30. Terminar( aceptar )
```

Aquí presentamos el algoritmo de *Recortar_Izquierda_Inferior()*:

```
booleano Recortar_Izquierda_Inferior( ref Punto P, ref Punto Q, Rectángulo R )
        1. Si Q.x < R.x_izq, entonces
                2. aceptar ← falso
        3. Si no, compruebe que Q.y < R.y_inf,
                4. aceptar ← falso
        5. Si no, entonces
                6. PQ ← Q - P
                7. PVis ← (R.x_izq,R.y_sup) - P
                8. Si Izquierda( PQ, PVis ), entonces
                        9. aceptar ← falso
                10. Si no, entonces
                        11. PVds ← (R.x_der,R.y_sup) - P
                        12. Si Izquierda( PQ, PVds ), entonces
                                13. aceptar ← verdadero
                                14. Si Q.y > R.y_sup, entonces
                                        15. Q ← Intersección_Horizontal( P, Q, R.y_sup )
                                16. PVii ← (R.x_izq,R.y_inf) - P
                                17. Si Izquierda( PQ, PVii ), entonces
                                        18. P ← Intersección_Vertical( P, Q, R.x_izq )
                                19. Si no, entonces
                                        20. P ← Intersección_Horizontal( P, Q, R.y_inf )
                        21. Si no, entonces
                                22. PVdi ← (R.x_der,R.y_inf) - P
                                23. Si Derecha( PQ, PVdi ), entonces
                                        24. aceptar ← falso
                                25. Si no, entonces,
                                        26. aceptar ← verdadero
                                        27. Si Q.x > R.x_der, entonces
                                                28. Q ← Intersección_Vertical( P, Q,
R.x_der )
                                        29. PVii ← (R.x_izq,R.y_inf) - P
                                        30. Si Izquierda( PQ, PVii ), entonces
                                                31. P ← Intersección_Vertical( P, Q,
R.x_izq )
                                        32. Si no, entonces
                                                33. P ← Intersección_Horizontal( P, Q,
R.y_inf )
        34.Terminar( aceptar )
```

Por último presentamos el algoritmo de *Recortar_Izquierda()*:

```
booleano Recortar_Izquierda( ref Punto P, ref Punto Q, Rectángulo R )
        1. Si Q.x < R.x_izq, entonces
                2. aceptar ← falso
        3. Si no, entonces
                4. PQ ← Q - P
                5. PVis ← (R.x_izq,R.y_sup) - P
                6. Si Izquierda( PQ, PVis ), entonces
                        7. aceptar ← falso
                8. Si no, entonces
                        9. PVds ← (R.x_der,R.y_sup) - P
                        10. Si Izquierda( PQ, PVds ), entonces
                                11. aceptar ← verdadero
                                12. P ← Intersección_Vertical( P, Q, R.x_izq )
                                13. Si Q.y > R.y_sup, entonces
                                        14. Q ← Intersección_Horizontal( P, Q, R.y_sup )
                        15. Si no, entonces
                                16. PVdi ← (R.x_der,R.y_inf) - P
                                17. Si Izquierda( PQ, PVdi ), entonces
                                        18. aceptar ← verdadero
                                        19. P ← Intersección_Vertical( P, Q, R.x_izq )
                                        20. Si Q.x > R.x_der, entonces
                                                21. Q ← Intersección_Vertical( P, Q,
R.x_der )
                                22. Si no, entonces
                                        23. PVii ← (R.x_izq,R.y_inf) - P
                                        24. Si Derecha( PQ, PVii ), entonces
                                                25. aceptar ← falso
                                        26. Si no, entonces
                                                27. aceptar ← verdadero
                                        28. P ← Intersección_Vertical( P, Q, R.x_izq )
                                        29. Si Q.y < R.y_inf, entonces
                                                30. Q ← Intersección_Horizontal( P, Q,
R.y_inf )
        31. Terminar( aceptar )
```

Estos cuatro algoritmos son auxiliares para los algoritmos de recorte.

```
booleano Izquierda( Vector A, Vector B )
        1. Si -B.y * A.x + B.x * A.y > 0, entonces
                2. Terminar( verdadero )
        3. Si no, entonces
                4. Terminar( falso )
```

```
booleano Derecha( Vector A, Vector B )
        1. Si -B.y * A.x + B.x * A.y < 0, entonces
                2. Terminar( verdadero )
        3. Si no, entonces
                4. Terminar( falso )
```

Podemos implementar este algoritmo como un caso especial de *Izquierda()*. Esto es,

```
booleano Derecha( Vector A, Vector B )
        1. Terminar( Izquierda( -A, B ) )
```

Por lo tanto, podríamos prescindir del uso de este algoritmo, *Derecha()*.

```
Punto Intersección_Vertical( Punto A, Punto B, real Arista )
        1. D ← B.x - A.x
        2. Si D = 0, entonces
                3. Resultado.x ← Arista
                4. Resultado.y ← A.y
        5. Si no, entonces
                6. Resultado.x ← Arista
                7. Resultado.y ← A.y - (B.y - A.y) * (A.x - Arista) / D
        8. Terminar( Resultado )
```

```
Punto Intersección_Horizontal( Punto A, Punto B, real Arista )
        1. D ← B.y - A.y
        2. Si D = 0, entonces
                3. Resultado.x ← A.x
                4. Resultado.y ← Arista
        5. Si no, entonces
                6. Resultado.x ← A.x - (B.x - A.x) * (A.y - Arista) / D
                7. Resultado.y ← Arista
        8. Terminar( Resultado )
```

Algoritmo de Sutherland-Hodgman

Este algoritmo se basa en recortar el polígono parcialmente usando cada lado del rectángulo vertical de recorte hasta terminar con la intersección final, interior al rectángulo. El polígono a recortar se representa como una lista de vértices: $v_1,v_2,v_3,...,v_n$. Nos interesa que esta lista de vértices esté ordenada para que cada pareja contigua de vértices represente una arista del polígono. Esto significa que ordenaremos la lista geométricamente en el sentido de las agujas del reloj o en el sentido contrario, según elijamos. Los vértices v_i y v_{i+1} describen una arista del polígono, hasta llegar a v_n que forma la última arista con el vértice, v_1. Cada recorte del polígono generará una nueva lista de vértices que representará el nuevo polígono resultante del recorte.

Para cada recorte por el lado del rectángulo, existen cuatro casos a comprobar para cada pareja de vértices que representa una arista del polígono a recortar. Según las regiones formadas por cada lado de recorte, los vértices se encontrarán dentro o fuera de ellas, éstos son:

1. Si ambos vértices están dentro, entonces sólo agregamos el segundo vértice a la lista resultante de vértices.

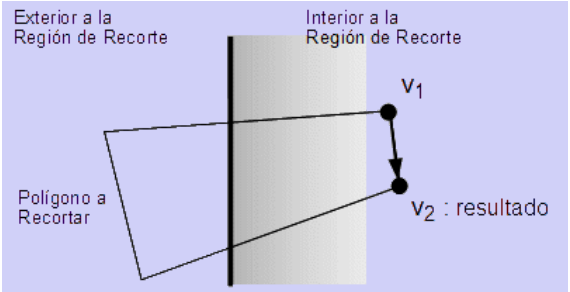


Figura 39 - Ambos vértices están dentro

2. Si el primer vértice está dentro de la arista, pero el segundo está fuera, entonces calculamos el punto de intersección. Sólo agregamos este punto de intersección a la lista resultante de vértices.

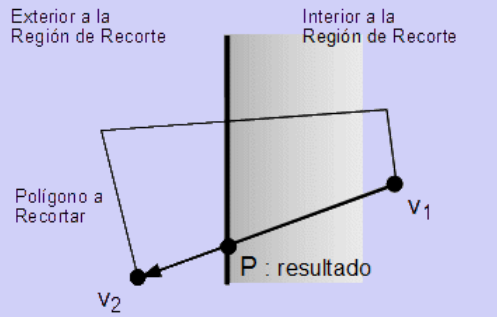


Figura 40 - El primer vértice está dentro, pero el segundo está fuera

3. Si el primer vértice está fuera de la arista, pero el segundo está dentro, entonces tenemos que calcular el punto de intersección. Este punto de intersección y el segundo vértice son agregados a la lista resultante de vértices.

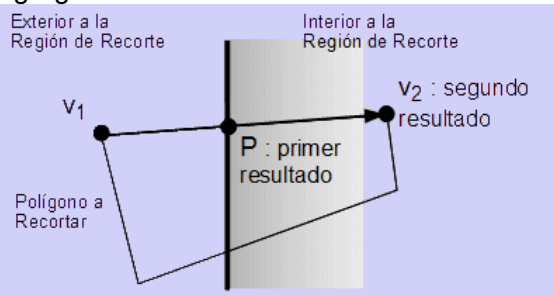


Figura 41 - El primer vértice está fuera, pero el segundo está dentro

4. Si ambos vértices están fuera, entonces no modificamos la lista resultante de vértices.

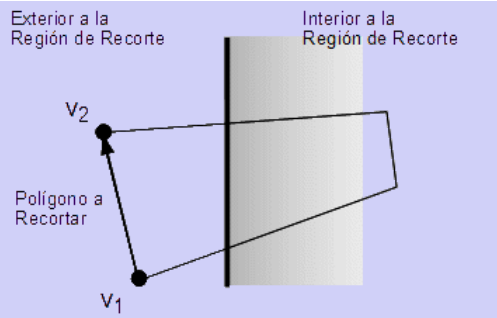


Figura 42 - Ambos vértices están fuera

Pasamos la lista resultante de vértices como la lista entrante al mismo algoritmo para las demás aristas del rectángulo.

Ejemplo

Descripción

Tenemos un rectángulo vertical de recorte cuya diagonal es de (-1,3) a (3,-3) que representa nuestra vista. Queremos un polígono (convexo), en tal rectángulo de recorte, descrito por la

siguiente lista de vértices:

```
{ (-2,1), (1,4), (4,3), (3,0),
(0,-4), (-2,-4), (-3,-1) }
```

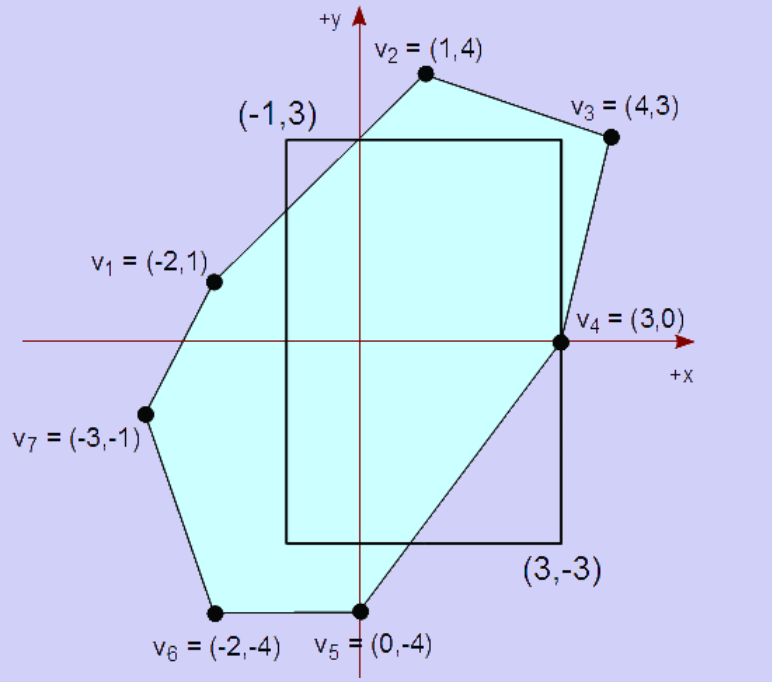


Figura 43 - Ejemplo del Algoritmo de Sutherland-Hodgman

Solución

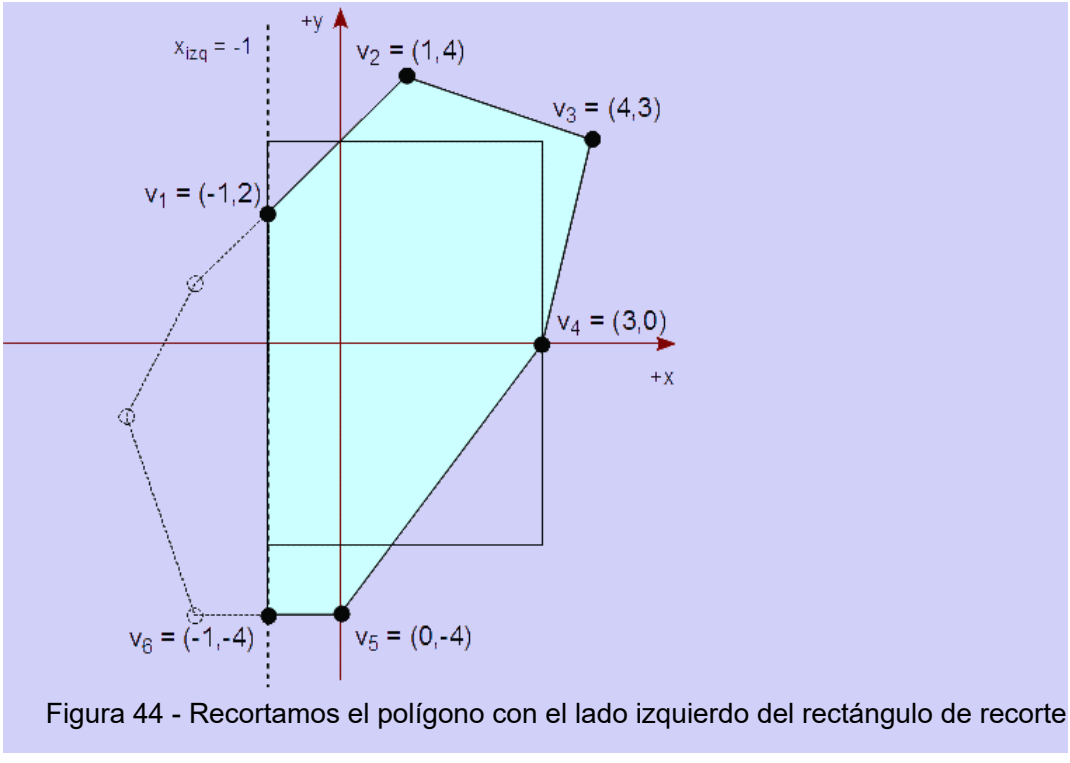


Figura 44 - Recortamos el polígono con el lado izquierdo del rectángulo de recorte

Empezamos el recorte de nuestro polígono con el lado izquierdo de nuestro rectángulo vertical de recorte que es $x_{izq} = -1$. Elegimos cada arista de nuestro polígono y la comprobamos según su caso, mencionado previamente. También creamos una lista de vértices para representar el polígono resultante del recorte.

Vemos que de v_1 a v_2, se nos presenta el caso #3, ya que v_1 está fuera del lado del rectángulo y v_2 está dentro. Por lo tanto, calculamos el punto de intersección del segmento con $x_{izq} = -1$,

$$P_x = x_{izq} = -1$$

$$P_y = v_{2_y} + (x_{izq} - v_{2_x}) * \frac{v_{2_y} - v_{1_y}}{v_{2_x} - v_{1_x}}$$

$$= 4 + (-1 - 1) * 3 / 3 = 2$$

Agregamos este punto de intersección y el vértice v_2 a la lista resultante de vértices, en este orden. La lista resultante es ahora:

```
{ (-1,2), (1,4) }
```

Comprobamos el segmento de v_2 a v_3. Como ambos son interiores al lado izquierdo del rectángulo de recorte, estamos ante el caso #1, por lo que agregamos v_3 a la lista resultante. La lista resultante es ahora:

```
{ (-1,2), (1,4), (4,3) }
```

Comprobamos el segmento de v_3 a v_4. Como ambos son interiores al lado izquierdo del rectángulo de recorte, estamos ante el caso #1, por lo que agregamos v_4 a la lista resultante. La lista resultante es ahora:

```
{ (-1,2), (1,4), (4,3), (3,0) }
```

Comprobamos el segmento de v_4 a v_5. Como ambos son interiores al lado izquierdo del rectángulo de recorte, estamos ante el caso #1, por lo que agregamos v_5 a la lista resultante. La lista resultante es ahora:

```
{ (-1,2), (1,4), (4,3), (3,0), (0,-4) }
```

Vemos que el segmento de v_5 a v_6 trata del caso #2. Por lo tanto, calculamos el punto de intersección del segmento con $x_{izq} = -1$,

$$P_x = x_{izq} = -1$$

$$P_y = v_{6_y} + (x_{izq} - v_{6_x}) * \frac{v_{6_y} - v_{5_y}}{v_{6_x} - v_{5_x}}$$

$$= -4 + (-1 - (-2)) * 0 / 2 = -4$$

Agregamos este punto de intersección a la lista resultante de vértices. La lista resultante es ahora:

```
{ (-1,2), (1,4), (4,3), (3,0), (0,-4), (-1,-4) }
```

Comprobando los dos últimos segmentos, v_6 a v_7 y v_7 a v_1, vemos que están fuera. Por lo tanto, no agregamos ningún vértice a la lista resultante. Al final, la lista resultante es:

```
{ (-1,2), (1,4), (4,3), (3,0), (0,-4), (-1,-4) }
```

Continuamos el recorte con el lado

superior de nuestro rectángulo vertical de recorte que es y_{sup} = 3. Elegimos cada arista de nuestro polígono actual, previamente recortado, y la comprobamos según su caso, mencionado previamente.

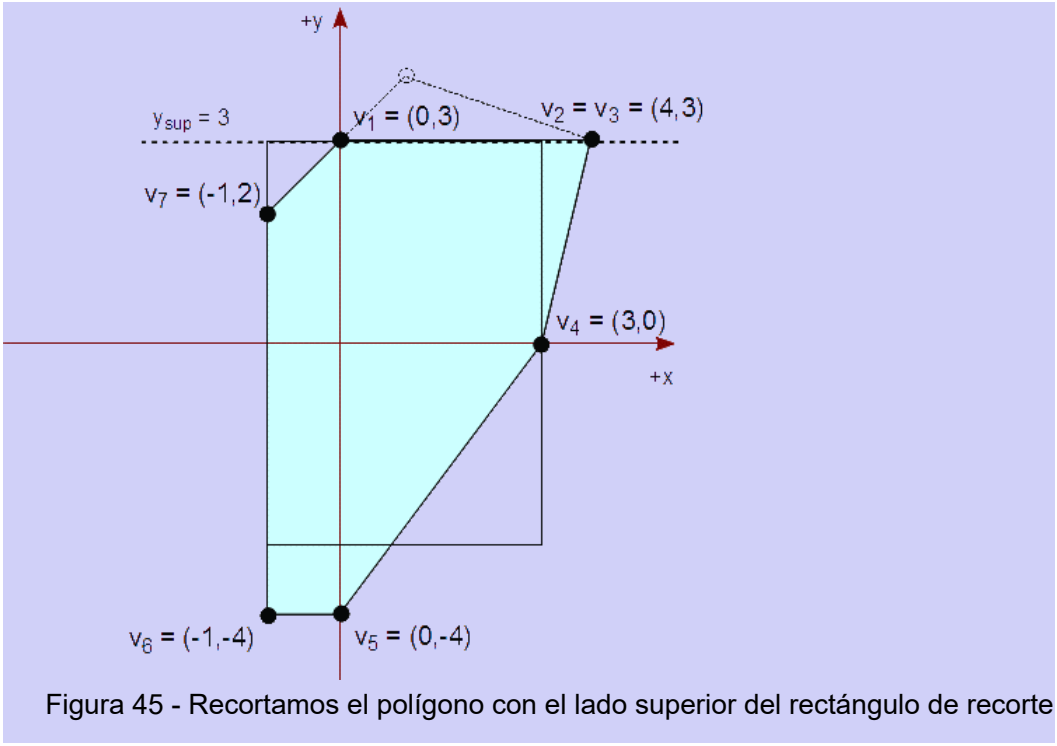


Figura 45 - Recortamos el polígono con el lado superior del rectángulo de recorte

Vemos que de v_1 a v_2, se nos presenta el caso #2, ya que v_1 está dentro del lado del rectángulo y v_2 está fuera. Por lo tanto, calculamos el punto de intersección del segmento con y_{sup} = 3,

$$P_x = v_{2_x} + (y_{sup} - v_{2_y}) * \frac{v_{2_x} - v_{1_x}}{v_{2_y} - v_{1_y}}$$
$$= 1 + (3 - 4) * 2 / 2 = 0$$
$$P_y = y_{sup} = 3$$

Agregamos este punto de intersección a la lista resultante de vértices. La lista resultante es ahora:

{ (0,3) }

Comprobamos que el segmento de v_2 a v_3 presenta el caso #3, ya que v_2 está fuera del lado del rectángulo y v_3 está (justo) dentro. Por lo tanto, calculamos el punto de intersección del segmento con y_{sup} = 3,

$$P_x = v_{3_x} + (y_{sup} - v_{3_y}) * \frac{v_{3_x} - v_{2_x}}{v_{3_y} - v_{2_y}}$$
$$= 4 + (3 - 3) * 3 / (-1) = 4$$
$$P_y = y_{sup} = 3$$

Agregamos este punto de intersección y el vértice v_3 a la lista resultante de vértices, en este orden. La lista resultante es ahora:

{ (0,3), (4,3), (4,3) }

Comprobando el resto de los segmentos vemos que están dentro del semiplano, $y_{sup} = 3$. Por lo tanto, agregamos los segundos vértices de cada segmento a la lista resultante. Al final, la lista resultante es:

{ (0,3), (4,3), (4,3), (3,0), (0,-4), (-1,-4), (-1,2) }

Continuamos el recorte con el lado derecho de nuestro rectángulo vertical de recorte que es $x_{der} = 3$. Elegimos cada arista de nuestro polígono actual, previamente recortado, y la comprobamos según su caso, mencionado previamente.

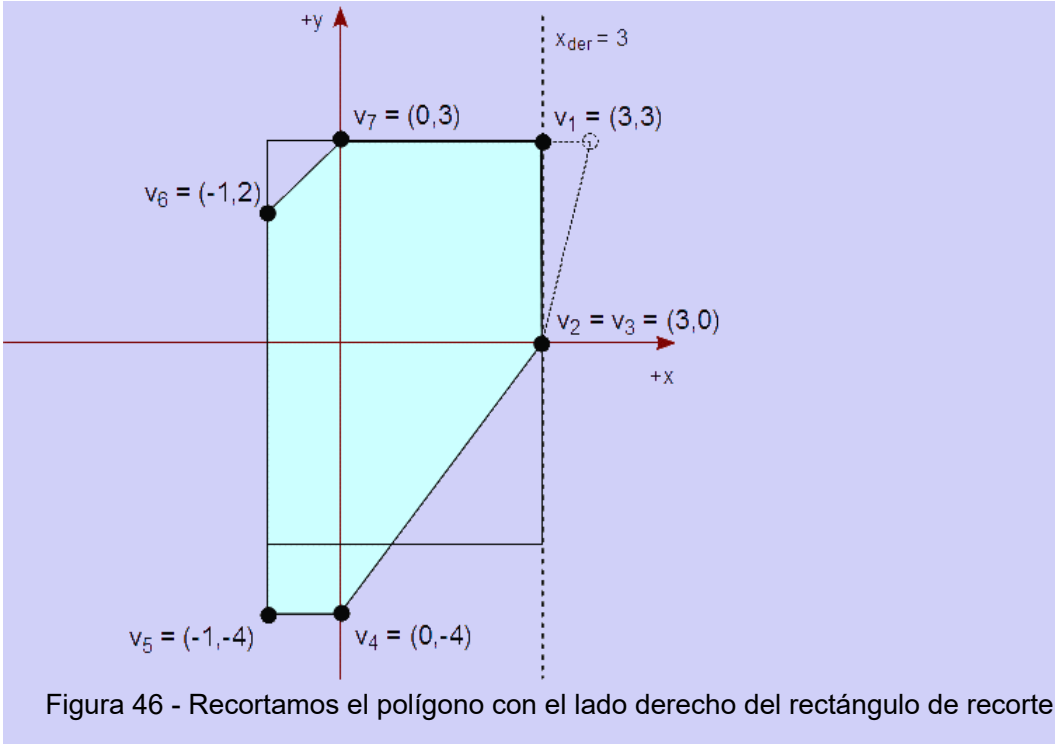


Figura 46 - Recortamos el polígono con el lado derecho del rectángulo de recorte

Vemos que de v_1 a v_2, se nos presenta el caso #2, ya que v_1 está dentro del lado del rectángulo y v_2 está fuera. Por lo tanto, calculamos el punto de intersección del segmento con $x_{der} = 3$,

$$P_x = x_{der} = 3$$

$$P_y = v_{2_y} + (x_{der} - v_{2_x}) * \frac{v_{2_y} - v_{1_y}}{v_{2_x} - v_{1_x}}$$

$$= 3 + (3 - 4) * 0 / 4 = 3$$

Agregamos este punto de intersección a la lista resultante de vértices. La lista resultante es ahora:

{ (3,3) }

Como el "segmento" de v_2 a v_3 está fuera del semiplano, $x_{der} = 3$, no agregamos ninguno de los dos vértices. La lista resultante sigue siendo:

{ (3,3) }

Comprobamos que el segmento de v_3 a v_4 presenta el caso #3, ya que v_3 está fuera del lado del rectángulo y v_4 está (justo) dentro. Por lo tanto, calculamos el punto de intersección del segmento con $x_{der} = 3$,

$$P_x = x_{der} = 3$$

$$P_y = v_{4_y} + (x_{der} - v_{4_x}) * \frac{v_{4_y} - v_{3_y}}{v_{4_x} - v_{3_x}}$$

$$= 0 + (3 - 3) * (-3) / (-1) = 0$$

Agregamos este punto de intersección y el vértice v_3 a la lista resultante de vértices, en este orden. La lista resultante es ahora:

```
{ (3,3), (3,0), (3,0) }
```

Comprobando el resto de los segmentos vemos que están dentro del semiplano, $x_{der} = 3$. Por lo tanto, agregamos los segundos vértices de cada segmento a la lista resultante. Al final, la lista resultante es:

```
{ (3,3), (3,0), (3,0), (0,-4), (-1,-4), (-1,2), (0,3) }
```

Continuamos el recorte con el lado inferior de nuestro rectángulo vertical de recorte que es $y_{inf} = -3$. Elegimos cada arista de nuestro polígono actual, previamente recortado, y la comprobamos según su caso, mencionado previamente.

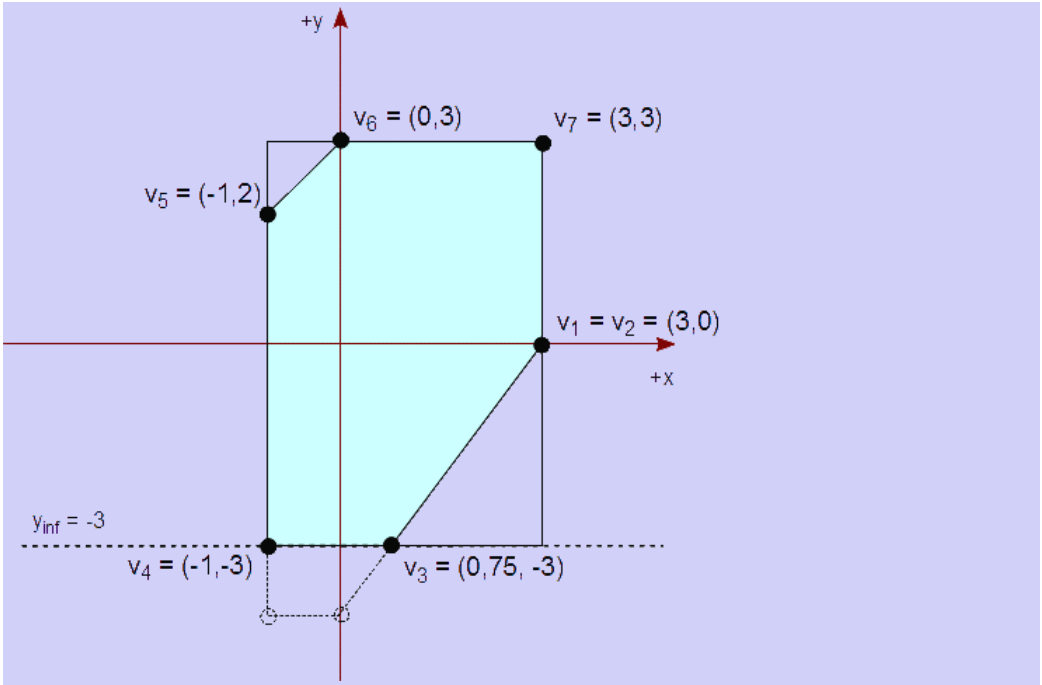


Figura 47 - Recortamos el polígono con el lado inferior del rectángulo de recorte

Comprobamos que tanto el segmento de v_1 a v_2 como el "segmento" de v_2 a v_3 están dentro del semiplano, $y_{inf} = -3$. Por lo tanto, agregamos los segundos vértices de cada segmento a la lista resultante. Al final, la lista resultante es:

```
{ (3,0), (3,0) }
```

Vemos que de v_3 a v_4, se nos presenta el caso #2, ya que v_3 está dentro del lado del rectángulo y v_4 está fuera. Por lo tanto, calculamos el punto de intersección del segmento con y_{inf} = -3,

$$P_x = v_{4_x} + (y_{inf} - v_{4_y}) * \frac{v_{4_x} - v_{3_x}}{v_{4_y} - v_{3_y}}$$
$$= 0 + (-3 - (-4)) * (-3) / (-4) = 0,75$$
$$P_y = y_{inf} = -3$$

Agregamos este punto de intersección a la lista resultante de vértices. La lista resultante es ahora:

```
{ (3,0), (3,0), (0,75, -3) }
```

Como el segmento de v_4 a v_5 está fuera del semiplano, y_{inf} = -3, no agregamos ninguno de los dos vértices. La lista resultante sigue siendo:

```
{ (3,0), (3,0), (0,75, -3) }
```

Comprobamos que el segmento de v_5 a v_6 presenta el caso #3, ya que v_5 está fuera del lado del rectángulo y v_6 está dentro. Por lo tanto, calculamos el punto de intersección del segmento con y_{inf} = -3,

$$P_x = v_{6_x} + (y_{inf} - v_{6_y}) * \frac{v_{6_x} - v_{5_x}}{v_{6_y} - v_{5_y}}$$
$$= -1 + (-3 - 2) * 0 / 6 = -1$$
$$P_y = y_{inf} = -3$$

Agregamos este punto de intersección y el segundo vértice, v_6, a la lista resultante de vértices. La lista resultante es ahora:

```
{ (3,0), (3,0), (0,75, -3), (-1,-3), (-1,2) }
```

Comprobando el resto de los segmentos vemos que están dentro del semiplano, y_{inf} = -3. Por lo tanto, agregamos los segundos vértices de cada segmento a la lista resultante. Al final, la lista resultante es:

```
{ (3,0), (3,0), (0,75, -3), (-1,-3), (-1,2), (0,3), (3,3) }
```

Vemos que el polígono final contiene vértices contiguos repetidos, por lo que podemos procesar nuestra lista de vértices para eliminar las repeticiones. Al final, nos queda la siguiente lista de vértices:

```
{ (3,0), (0,75, -3),
(-1,-3), (-1,2), (0,3),
(3,3) }
```

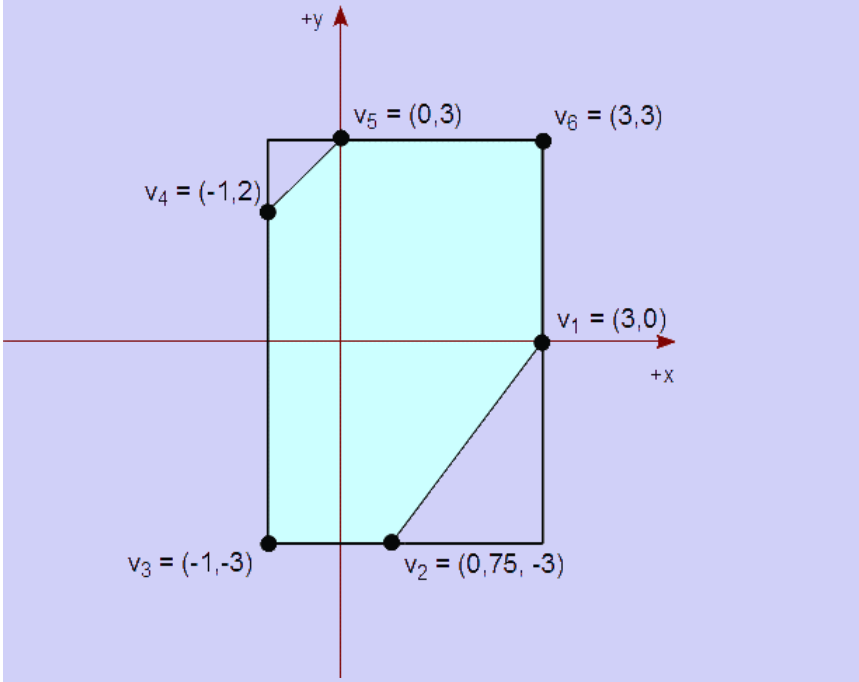


Figura 48 - El polígono resultante sin vértices contiguos repetidos

Algoritmo

El algoritmo se basa en el pseudo-código principal, *Recortar()*. Tenemos otros algoritmos auxiliares:

- *Recortar_Lado()* sirve para recortar un segmento descrito por dos vértices por un lado particular del rectángulo de recorte. También se requiere la lista resultante de vértices para poder agregar nuevos vértices a ella. La salida es la misma lista resultante actualizada de los vértices del polígono recortado.
- *Interior()* indica si un punto se considera candidato como interior (verdadero) al rectángulo de recorte o se acepta como exterior (falso).
- *Intersectar()* calcula el punto de intersección entre un segmento y un lado dado del rectángulo de recorte.

Presentamos el algoritmo para *Recortar()*:

```
Polígono Recortar( Polígono P, Rectángulo R )
        1. Resultado, Actual : Polígono
        2. Actual ← P
        3. Para cada lado de R, repetir
                4. Resultado ← vacío  // Resultado es un polígono vacío
                5. Para: i ← 1 hasta n, repetir
                        6. Resultado ← Recortar_Lado( Actual.v_i, Actual.v_i+1, lado, R,
Resultado )
                7. Resultado ← Recortar_Lado( Actual.v_n, Actual.v_1, lado, R, Resultado )
// Recortamos la última arista de Actual: v_n v_1
                8. Actual ← Resultado
        9. Resultado ← Eliminar_Repeticiones( Resultado )
        10. Terminar( Resultado )
```

Eliminar_Repeticiones() recorre la lista dada de vértices del polígono para eliminar repeticiones contiguas. También hay que comprobar el último vértice con el primero, ya que forman un segmento.

He aquí el algoritmo de *Recortar_Lado()* que modifica el polígono, *Resultado*, y lo regresa al terminar:

```
Polígono Recortar_Lado( Punto v1, Punto v2, Lado L, Rectángulo R, ref Polígono Resultado
)
        1. interior1 ← Interior( v1, L, R )
        2. interior2 ← Interior( v2, L, R )
        3. Si interior1 = verdadero, entonces
                4. Si interior2 = verdadero, entonces
                        5. Resultado.Agregar( v2 )
                6. Si no, entonces,
                        7. P ← Intersectar( v1, v2, L, R )
                        8. Resultado.Agregar( P )
        9. Si no, compruebe que  interior2 = verdadero,
                10. P ← Intersectar( v1, v2, L, R )
                11. Resultado.Agregar( P )
                12. Resultado.Agregar( v2 )
        13. Terminar( Resultado )
```

Agregar() se refiere a la operación básica para agregar un punto a la lista de vértices de un polígono.

Exponemos la lógica de *Interior()*:

```
booleano Interior( Punto v, Lado L, Rectángulo R )
        1. Si L = Izquierdo, entonces
                2. Si v_x ≥ R.x_izq, entonces
                        3. bEsInterior ← verdadero
                4. Si no, entonces
                        5. bEsInterior ← falso
        6. Si no, compruebe que  L = Superior, entonces
                7. Si v_y ≤ R.y_sup, entonces
                        8. bEsInterior ← verdadero
                9. Si no, entonces
                        10. bEsInterior ← falso
        11. Si no, compruebe que  L = Derecho, entonces
                12. Si v_x ≤ R.x_der, entonces
                        13. bEsInterior ← verdadero
                14. Si no, entonces
                        15. bEsInterior ← falso
        16. Si no, compruebe que  L = Inferior, entonces
                17. Si v_y ≥ R.y_inf, entonces
                        18. bEsInterior ← verdadero
                19. Si no, entonces
                        20. bEsInterior ← falso
        21. Terminar( bEsInterior )
```

Este algoritmo para *Intersectar()* involucra un rectángulo vertical de recorte:

```
Punto Intersectar( Punto v1, Punto v2, Lado L, Rectángulo R )
        1. dx ← v2.x - v1.x
        2. dy ← v2.y - v1.y
        3. Si L = Izquierdo, entonces
                4. P.x ← R.x_izq
                5. Si dx ≠ 0, entonces
```

```
                6. P.y ← v2.y + (R.x_izq - v2.x) * dy / dx
        7. Si no, entonces
                8. P.y ← v2.y
9. Si no, compruebe que  L = Superior, entonces
        10. Si dy ≠ 0, entonces
                11. P.x ← v2.x + (R.y_sup - v2.y) * dx / dy
        12. Si no, entonces
                13. P.x ← v2.x
        14. P.y ← R.y_sup
15. Si no, compruebe que  L = Derecho, entonces
        16. P.x ← R.x_der
        17. Si dx ≠ 0, entonces
                18. P.y ← v2.y + (R.x_der - v2.x) * dy / dx
        19. Si no, entonces
                20. P.y ← v2.y
21. Si no, compruebe que  L = Inferior, entonces
        22. Si dy ≠ 0, entonces
                23. P.x ← v2.x + (R.y_inf - v2.y) * dx / dy
        24. Si no, entonces
                25. P.x ← v2.x
        26. P.y ← R.y_inf
27. Terminar( P )
```

Algoritmo de Liang-Barsky

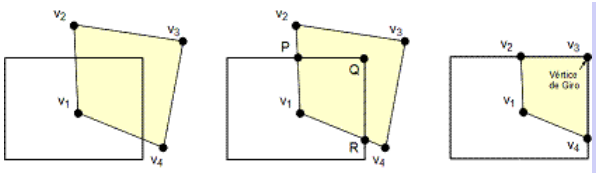


Figura 49 - Se agrega el vértice de giro, que es un vértice del rectángulo de recorte

El algoritmo de Liang-Barsky para recortar líneas puede ser ampliado para recortar polígonos en un rectángulo vertical. La idea principal de este algoritmo se basa en la detección de *vértices de giro*. Un vértice de giro es aquel vértice de la esquina del rectángulo de recorte que formará parte del polígono recortado. Esta posibilidad existe si una arista del polígono a recortar cruza un lado del rectángulo de recorte seguida de una o más aristas que giran entorno a la esquina del rectángulo de recorte, para volver a cruzar el rectángulo por otro lado. Si esto ocurriere, entonces agregaríamos este vértice de giro a nuestra lista de vértices resultante que representa el polígono recortado.

Fijémonos en varios casos de intersección de líneas rectas diagonales - aquéllas que no son verticales ni horizontales. Extendemos las aristas del rectángulo vertical para que sean líneas y así creamos nueve regiones: ocho externas y una interna. Esto implica que una línea diagonal cruza de una esquina regional a otra esquina opuesta.

Si parte de un segmento del polígono yace dentro del rectángulo de recorte, entonces esa parte debe pertenecer al polígono resultante. Debemos agregar los vértices de este segmento, dependiendo de las circunstancias:

- El segmento yace completamente en el interior del rectángulo de recorte, por lo que ambos extremos del segmento son agregados a la lista de vértices del polígono resultante.
- El segmento yace parcialmente en el interior, con un extremo dentro del rectángulo de recorte. Esto implica que hay que calcular el punto de intersección con un lado del rectángulo. Se agrega el extremo interior del segmento y el punto de intersección a la lista de vértices del polígono resultante.

- El segmento yace parcialmente en el interior, pero ambos puntos extremos yacen fuera del rectángulo de recorte. Esto supone calcular dos puntos de intersección con dos lados diferentes del rectángulo. Ambos puntos de intersección son agregados a la lista de vértices del polígono resultante.

Por otro lado, podemos tener el caso de que el segmento no cruce el interior del rectángulo de recorte, pero el siguiente segmento que representa una arista del polígono sí puede cruzar el rectángulo. Como nos limitamos a mirar cada arista según su primer vértice, podemos determinar el lado del rectángulo, si se empieza en una región adyacente a un lado, que el segmento puede cruzar, o entre dos lados, si se empieza en una esquina.

Volviendo al tema del vértice de giro, agregamos tal vértice a la lista de vértices resultante, cuando una arista entra una esquina.

El algoritmo original propuesto por Liang y Barsky funciona de otra manera. Se agrega el vértice de giro a la lista resultante cuando una arista abandona la esquina. Sin embargo, esto no elimina el problema de la arista degenerada y además complica la lectura del algoritmo más de la cuenta. Por lo tanto, usaremos nuestra versión del algoritmo.

Basándonos en la solución dada por Liang y Barsky para recortar segmentos, haremos uso de la ecuación paramétrica de una línea recta y analizaremos los cuatro valores del parámetro, t, debidos a las cuatro intersecciones de la línea, que contiene la arista, con las líneas formadas por las extensiones de los lados del rectángulo de recorte. Dos valores son considerados potencialmente entrantes: t_{e_1} y t_{e_2} y los otros dos son considerados potencialmente salientes: t_{s_1} y t_{s_2}. Tomemos nota de que t_{e_1} es el menor y t_{s_2} es el mayor de los cuatro valores calculados; los otros dos valores pueden estar en cualquier orden. Como explicamos en el apartado del algoritmo de Liang-Barsky para recortar segmentos, si $t_{e_2} \leq t_{s_1}$, entonces la línea cruza el rectángulo de recorte; si $t_{e_2} > t_{s_1}$, entonces la línea pasa de una esquina a otra.

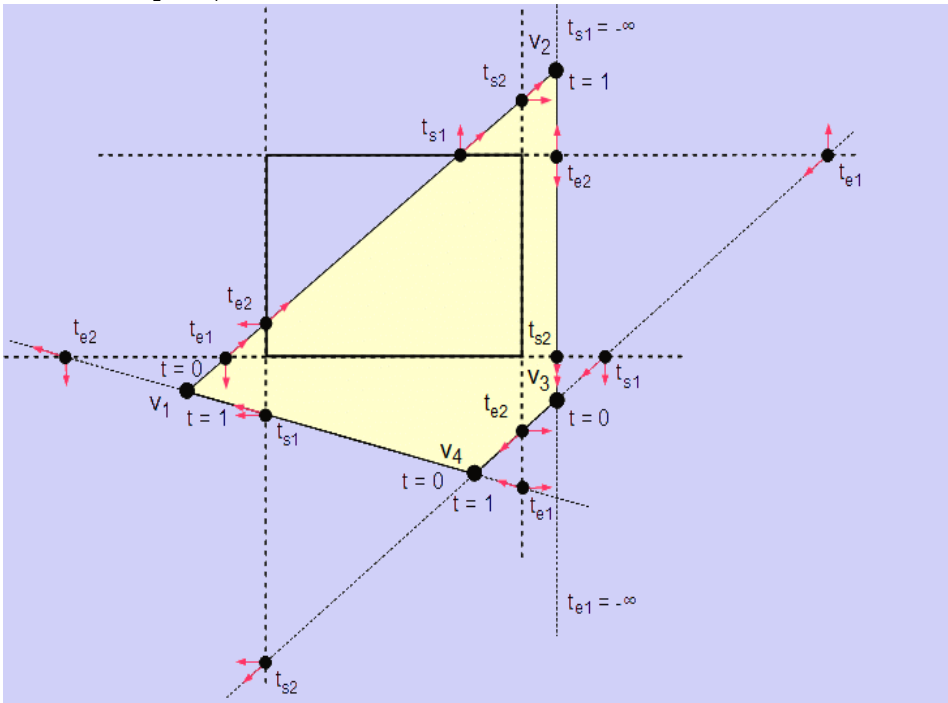


Figura 50 - Diferentes casos de los cuatro parámetros de puntos de intersección

Como ya sabemos, los valores de *t* deben estar comprendidos entre 0 y 1 para representar el segmento, donde t=0 implica un punto extremo y t=1 se refiere al otro. Estableciendo la relación entre estos valores paramétricos y los cuatro valores de las intersecciones, podemos determinar la contribución de la arista al polígono recortado resultante. Si $0 < t_{s_1}$ y $t_{e_2} \leq 1$, entonces la arista comienza o incluso puede terminar dentro del rectángulo. Como existe una parte visible de esta arista, debemos agregarla a nuestro polígono recortado.

Si la arista no cruza el rectángulo de recorte, entonces la línea, donde yace la arista, atraviesa tres esquinas: dos opuestas entre sí y una en el medio. Si la arista yace en cualquiera de estas dos esquinas opuestas, entonces se debe agregar un vértice de giro a la lista de vértices resultante. Entrada a esta esquina del medio, se comprueba con $0 < t_{s_1} \leq 1$. Entrando la última esquina se comprueba con $0 < t_{s_2} \leq 1$, que también es válida para el caso de que la arista cruce el rectángulo de recorte, y por tanto se debe agregar el vértice de giro.

El problema del algoritmo es que tenemos que tratar los casos especiales cuando las aristas son horizontales o verticales. Una solución es considerar cada caso cuando se presente, aplicando otra lógica especial que requiere un análisis para cada caso. La otra solución es describir estas aristas de tal manera que podamos usar la misma lógica del algoritmo para todas las aristas. Para implementar esta solución, asignamos los valores de +∞ y -∞ para los parámetros entrantes y salientes.

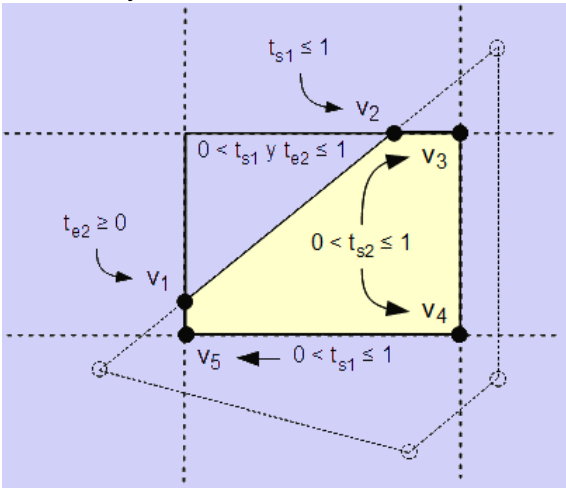


Figura 51 - Polígono recortado junto con las condiciones satisfechas

Ejemplo

Descripción

Retomamos el mismo ejemplo anterior del método de Sutherland-Hodgman. Tenemos un rectángulo vertical de recorte cuya diagonal es de (-1,3) a (3,-3) que representa nuestra vista. Queremos un polígono, en tal rectángulo de recorte, a partir del polígono, *P*, descrito por la siguiente lista de vértices:

```
P : { (-2,1), (1,4), (4,3), (3,0), (0,-4), (-2,-4), (-3,-1) }
```

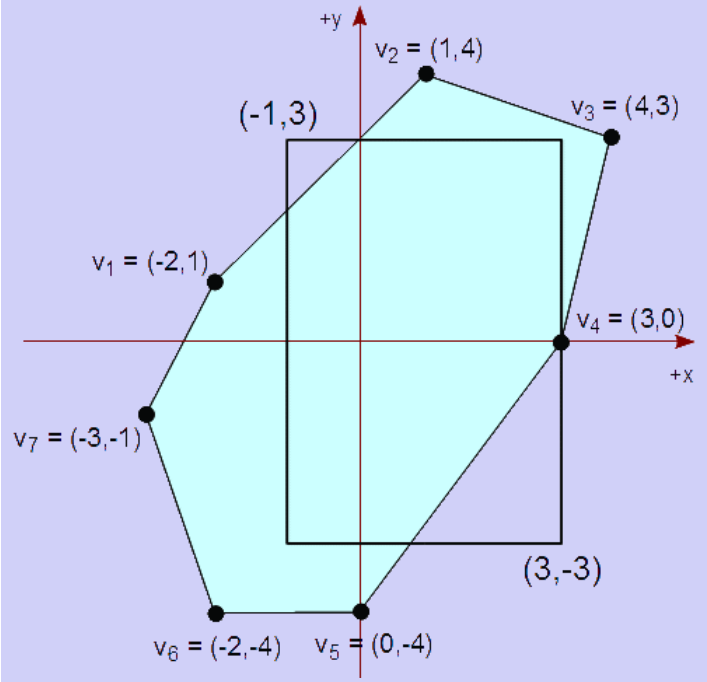


Figura 52 - Ejemplo del Algoritmo de Liang-Barsky

Solución

Creamos una lista de vértices, Q, para representar el polígono resultante del recorte, inicialmente vacía:

```
Q : {}
```

Empezamos el recorte de nuestro polígono analizando el primer segmento formado por v_1v_2. Calculamos las cuatro intersecciones y las clasificamos a través de sus parámetros.

$\Delta_x = v_{2_x} - v_{1_x} = 1 - (-2) = 3$
$\Delta_y = v_{2_y} - v_{1_y} = 4 - 1 = 3$

$t_{e_1} = (v_{1_y} - y_{inf}) / -\Delta_y = (1 - (-3)) / (-3) = -1,3333$
$t_{e_2} = -(v_{1_x} - x_{izq}) / \Delta_x = - ((-2) - (-1)) / 3 = 0,3333$
$t_{s_1} = -(v_{1_y} - y_{sup}) / \Delta_y = - (1 - 3) / 3 =$

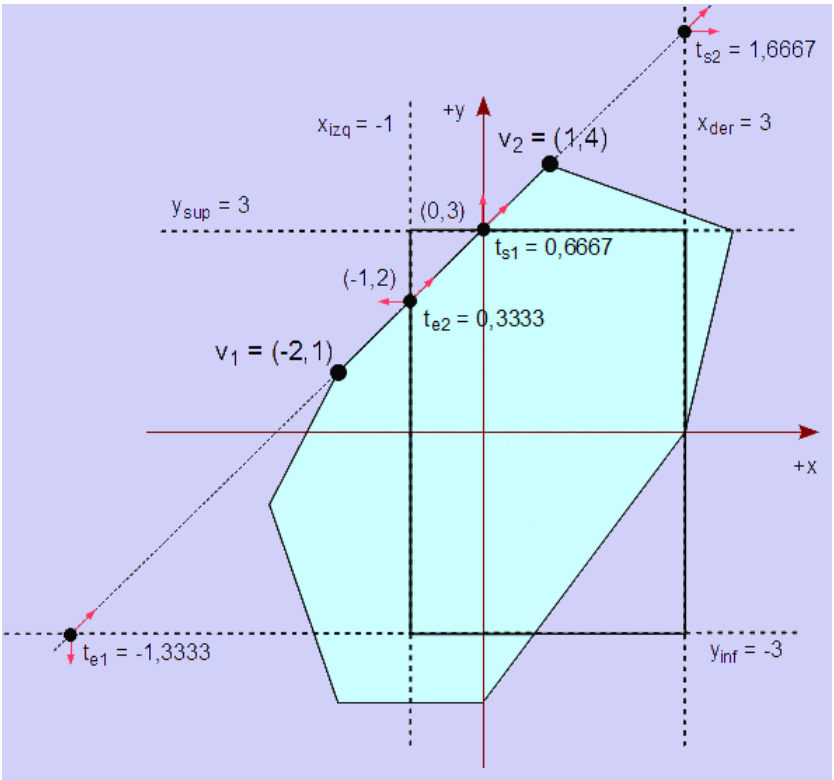


Figura 53 - Analizamos la primera arista del polígono

$0,6667$

$t_{s_2} = (v_{1_x} - x_{der}) / -\Delta_x = ((-2) - 3) / (-3) = 1,6667$

Como $t_{s_2} > 0$, $t_{e_2} \leq t_{s_1}$, $0 < t_{s_1}$ y $t_{e_2} \leq 1$, el segmento es completa o parcialmente visible en el rectángulo de recorte. Como $t_{e_2} > 0$, calculamos el punto de intersección

$(x,y) = v_1 + t_{e_2} * (\Delta_x,\Delta_y) = (-2,1) + 0,3333 * (3,3)$
$= (-1,2)$

y lo agregamos a la lista resultante, *Q*. La lista resultante es ahora:

$Q : \{ (-1,2) \}$

Como $t_{s_1} < 1$, calculamos el punto de intersección

$(x,y) = v_{1_x} + t_{s_1} * (\Delta_x,\Delta_y) = (-2,1) + 0,6667 * (3,3)$
$= (0,3)$

y lo agregamos a la lista resultante, *Q*. La lista resultante es ahora:

$Q : \{ (-1,2), (0,3) \}$

Seguimos con el análisis con el segundo segmento formado por v_2v_3. Calculamos las cuatro intersecciones y las clasificamos a través de sus parámetros.

$\Delta_x = v_{3_x} - v_{2_x} = 4 - 1 = 3$
$\Delta_y = v_{3_y} - v_{2_y} = 3 - 4 = -1$

$t_{e_1} = -(v_{2_x} - x_{izq}) / \Delta_x = -(1 - (-1)) / 3 = -0,6667$
$t_{e_2} = -(v_{2_y} - y_{sup}) / \Delta_y = -(4 - 3) / (-1) = 1$
$t_{s_1} = (v_{2_x} - x_{der}) / -\Delta_x = (1 - 3) / (-3) = 0,6667$
$t_{s_2} = (v_{2_y} - y_{inf}) / -\Delta_y = (4 - (-3)) / -(-1) = 7$

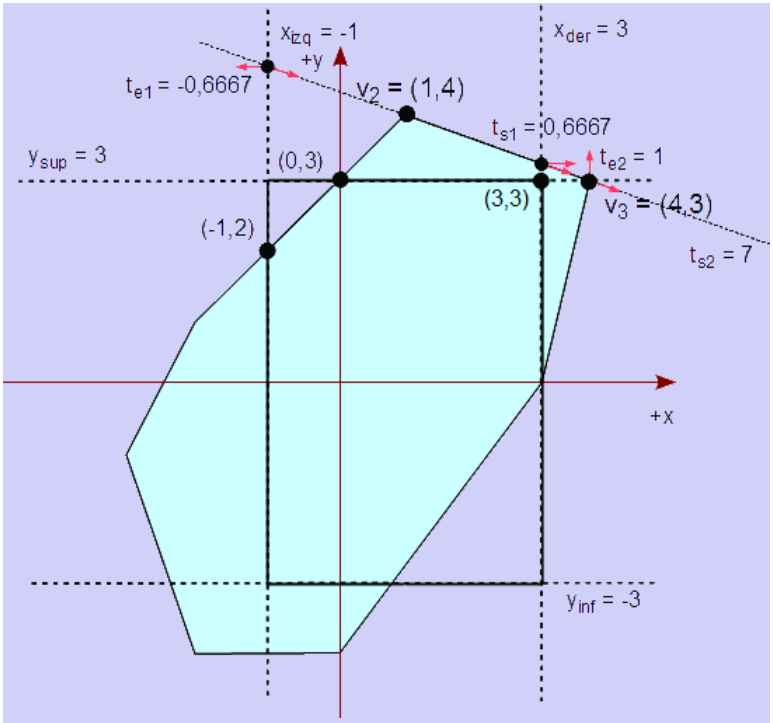


Figura 54 - Analizamos la segunda arista del polígono

Aunque $t_{s_2} > 0$, $t_{e_2} > t_{s_1}$, el segmento no es visible en el rectángulo de recorte. Sin embargo, como $0 < t_{s_1} \leq 1$, agregamos el vértice de giro: (3,3) a la lista resultante, *Q*. La lista resultante es ahora:

$Q : \{ (-1,2), (3,3) \}$

Como $t_{s_2} > 1$, no agregamos el otro vértice de giro: (3,-3).

Analicemos el tercer segmento formado por v_3v_4. Calculamos las

cuatro intersecciones y las clasificamos a través de sus parámetros.

$\Delta_x = v_{4_x} - v_{3_x} = 3 - 4 = -1$
$\Delta_y = v_{4_y} - v_{3_y} = 0 - 3 = -3$

$t_{e_1} = -(v_{3_y} - y_{sup}) / \Delta_y = -(3 - 3) / (-3) = 0$
$t_{e_2} = (v_{3_x} - x_{der}) / -\Delta_x = (4 - 3) / -(-1) = 1$
$t_{s_1} = (v_{3_y} - y_{inf}) / -\Delta_y = (3 - (-3)) / -(-3) = 2$
$t_{s_2} = -(v_{3_x} - x_{izq}) / \Delta_x = -(4 - (-1)) / (-1) = 5$

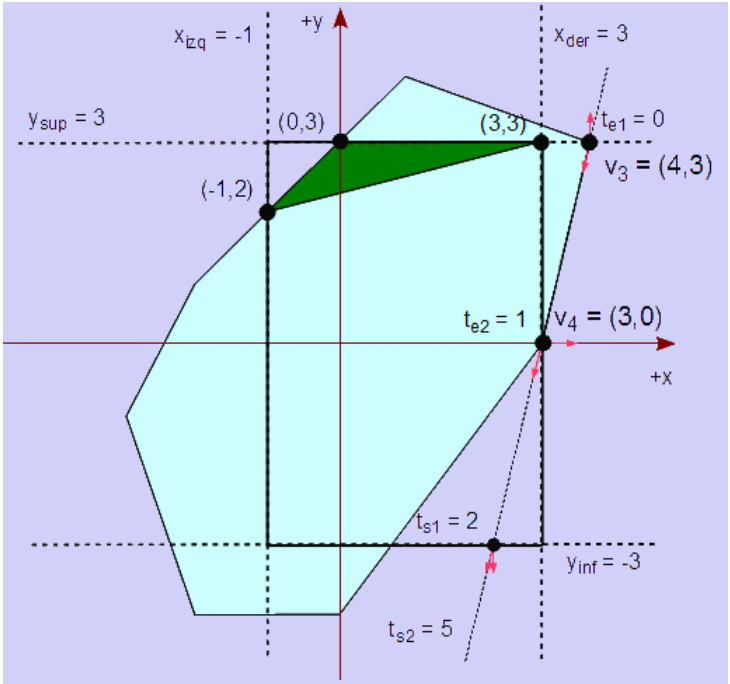


Figura 55 - Analizamos la tercera arista del polígono

Como $t_{s_2} > 0$, $t_{e_2} \leq t_{s_1}$, $0 < t_{s_1}$ y $t_{e_2} \leq 1$, el segmento es completa o parcialmente visible en el rectángulo de recorte. Como $t_{e_2} > 0$, calculamos el punto de intersección:

$(x,y) = v_3 + t_{e_2} * (\Delta_x,\Delta_y) = (4,3) + 1 * (-1,-3) = (3,0)$

y lo agregamos a la lista resultante, *Q*. La lista resultante es ahora:

Q : { (-1,2), (3,3), (3,0) }

Como $t_{s_1} \geq 1$, agregamos el vértice final del segmento, (3,0), a la lista resultante que es ahora:

Q : { (-1,2), (3,3), (3,0), (3,0) }

Como $t_{s_2} \geq 1$, no agregamos un vértice de giro.

Analicemos el cuarto segmento formado por v_4v_5. Calculamos las cuatro intersecciones y las clasificamos a través de sus parámetros.

$\Delta_x = v_{5_x} - v_{4_x} = 0 - 3 = -3$
$\Delta_y = v_{5_y} - v_{4_y} = -4 - 0 = -4$

$t_{e_1} = -(v_{4_y} - y_{sup}) / \Delta_y = -(0 - 3) / (-4) = -0,75$
$t_{e_2} = (v_{4_x} - x_{der}) / -\Delta_x = (3 - 3) / -(-3) = 0$
$t_{s_1} = (v_{4_y} - y_{inf}) / -\Delta_y = (0 - (-3)) / -(-4) = 0,75$
$t_{s_2} = -(v_{4_x} - x_{izq}) / \Delta_x = -$

```
( 3 - (-1) ) / (-3)  =
1,3333
```

Como $t_{s_2} > 0$, $t_{e_2} \leq t_{s_1}$, $0 < t_{s_1}$ y $t_{e_2} \leq 1$, el segmento es completa o parcialmente visible en el rectángulo de recorte. Como $t_{e_2} \leq 0$, agregamos el vértice inicial del segmento, (3,0), a la lista resultante que es ahora:

```
Q : { (-1,2), (3,3), (3,0),
(3,0), (3,0) }
```

Como $t_{s_1} < 1$, calculamos la intersección:

```
(x,y) = v4 + ts1 * (Δx,Δy) =
(3,0) + 0,75 * (-3,-4)
      = (0,75, -3)
```

y lo agregamos a la lista resultante, Q. La lista resultante es ahora:

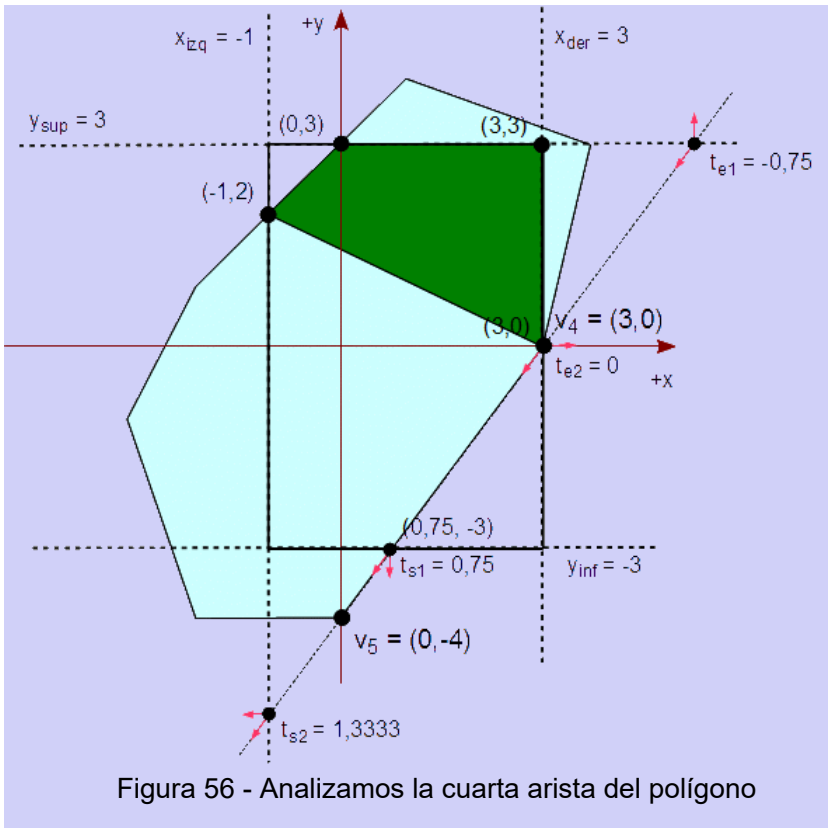


Figura 56 - Analizamos la cuarta arista del polígono

```
Q : { (-1,2), (3,3), (3,0), (3,0), (3,0), (0,75, -3) }
```

Como $t_{s_2} \geq 1$, no agregamos un vértice de giro.

Analicemos el quinto segmento formado por v_5v_6. Calculamos las cuatro intersecciones y las clasificamos a través de sus parámetros.

```
Δx = v6x - v5x = -2 - 0 = -2
Δy = v6y - v5y = -4 - (-4) = 0

te1 = -∞
te2 =  (v5x - xder) / -Δx =  ( 0 - 3 ) /
-(-2)    = -1,5
ts1 = -∞
ts2 = -(v5x - xizq) /  Δx = -( 0 - (-1)
) / (-2)  =  0,5
```

Aunque $t_{s_2} > 0$, $t_{s_1} < t_{e_2}$ y por tanto, el segmento no es visible en el rectángulo de recorte. Como $0 < t_{s_2} \leq 1$, agregamos el vértice de giro, (-1,-3) a la lista resultante, Q. La lista resultante es ahora:

```
Q : { (-1,2), (3,3), (3,0), (3,0),
(3,0), (0,75, -3), (-1,-3) }
```

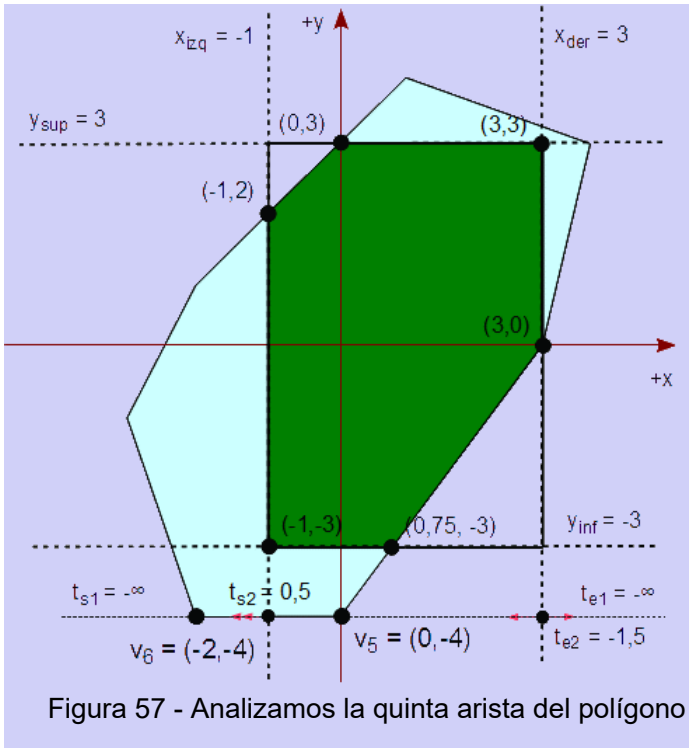


Figura 57 - Analizamos la quinta arista del polígono

Analicemos el sexto segmento formado por v_6v_7. Calculamos las cuatro intersecciones y las clasificamos a través de sus parámetros.

$\Delta_x = v_{7_x} - v_{6_x} = -3 - (-2) = -1$
$\Delta_y = v_{7_y} - v_{6_y} = -1 - (-4) = 3$

$t_{e_1} = (v_{6_x} - x_{der}) / -\Delta_x = (-2 - 3) / -(-1) = -5$
$t_{e_2} = (v_{6_y} - y_{inf}) / -\Delta_y = (-4 - (-3)) / (-3) = 0,3333$
$t_{s_1} = -(v_{6_x} - x_{izq}) / \Delta_x = -(-2 - (-1)) / (-1) = -1$
$t_{s_2} = -(v_{6_y} - y_{sup}) / \Delta_y = -(-4 - 3) / 3 = 2,3333$

Aunque $t_{s_2} > 0$, $t_{s_1} < t_{e_2}$ y por tanto, el segmento no es visible en el rectángulo de recorte. Como $t_{s_2} > 1$, no agregamos un vértice de giro. La lista resultante sigue siendo:

Q : { (-1,2), (3,3), (3,0), (3,0), (3,0), (0,75, -3), (-1,-3) }

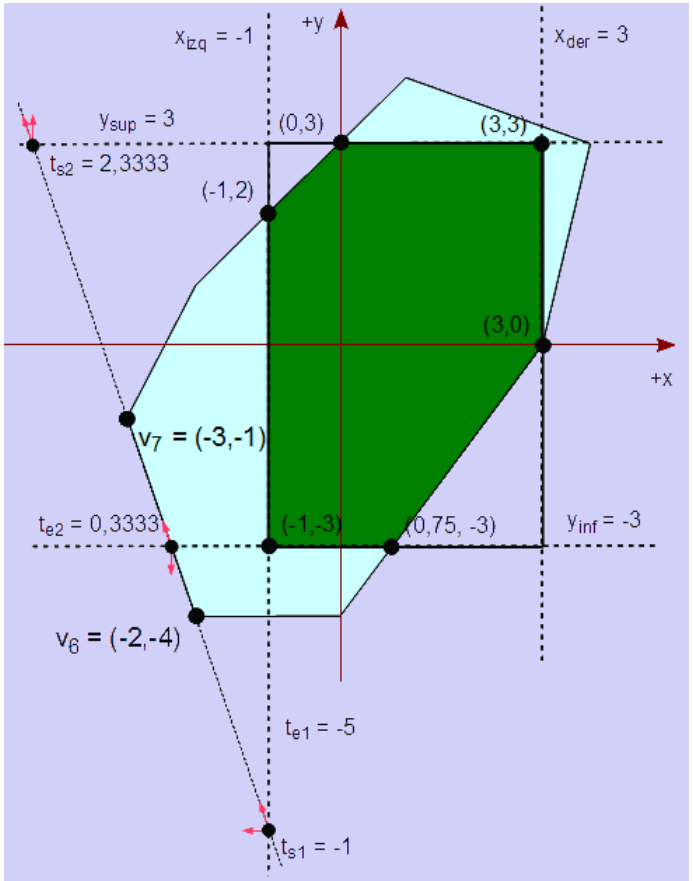


Figura 58 - Analizamos la sexta arista del polígono

Analicemos el último segmento formado por v_7v_1. Calculamos las cuatro intersecciones y las clasificamos a través de sus parámetros.

$\Delta_x = v_{1_x} - v_{7_x} = -2 - (-3) = 1$
$\Delta_y = v_{1_y} - v_{7_y} = 1 - (-1) = 2$

$t_{e_1} = (v_{7_y} - y_{inf}) / -\Delta_y = (-1 - (-3)) / (-2) = -2$
$t_{e_2} = -(v_{7_x} - x_{izq}) / \Delta_x = -(-3 - (-1)) / 1 = 2$
$t_{s_1} = -(v_{7_y} - y_{sup}) / \Delta_y = -(-1 - 3) / 2 = 2$
$t_{s_2} = (v_{7_x} - x_{der}) / -\Delta_x = (-3 - 3) / (-1) = 6$

Aunque $t_{s_2} > 0$, $t_{e_2} \geq t_{s_1}$, y $0 < t_{s_1}$, el segmento no es visible en el rectángulo de recorte ni se agrega ninguna intersección, porque $t_{e_2} > 1$. Como $t_{s_2} > 1$, tampoco agregamos un vértice de giro. La lista resultante sigue siendo:

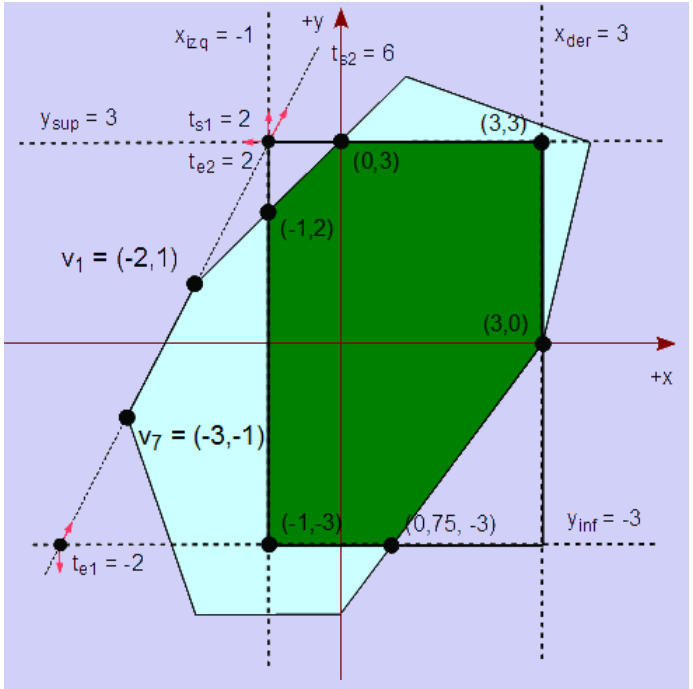


Figura 59 - Analizamos la última arista del polígono

```
Q : { (-1,2), (3,3), (3,0), (3,0), (3,0), (0,75, -3), (-1,-3) }
```

Como existen puntos contiguos repetidos en la lista de vértices del polígono resultante, nos interesaría eliminarlos. Al final, la lista resultante es:

```
{ (-1,2), (3,3), (3,0), (0,75, -3),
(-1,-3) }
```

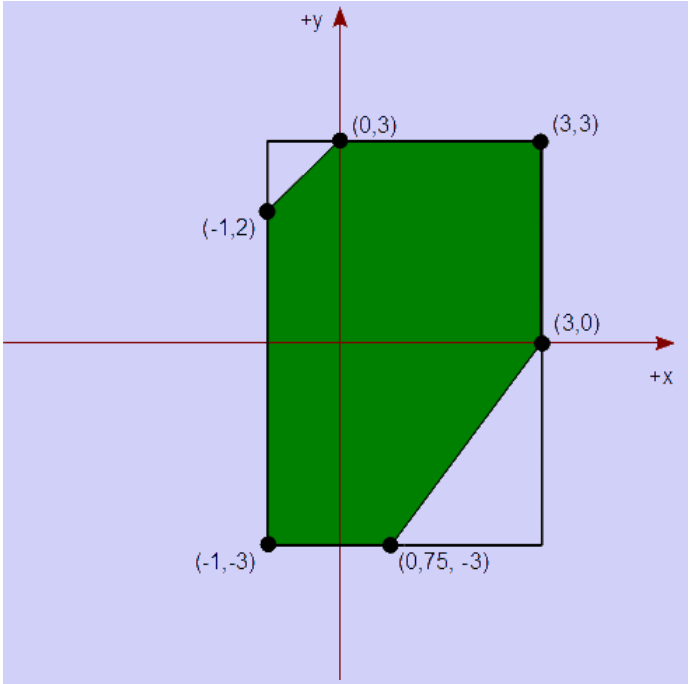


Figura 60 - El polígono resultante

Algoritmo

Presentamos el algoritmo detallado y optimizado para *Recortar()* basado en el algoritmo presentado por Foley en su libro:

```
Polígono Recortar( Polígono P, Rectángulo R )
        1. Crear Polígono: Resultado ← vacío
        2. P.Agregar( P.v[1].x, P.v[1].y )  // Agregamos el primer vértice al final
        3. Para: i ← 1 hasta P.cantidad, repetir
                4. dx ← P.v[i+1].x - P.v[i].x
                5. dy ← P.v[i+1].y - P.v[i].y
                6. Si dx > 0, o Si dx = 0 y P.v[i].x > R.x_der, entonces
                        7. x_e ← R.x_izq
                        8. x_s ← R.x_der
                9. Si no, entonces
                        10. x_e ← R.x_der
                        11. x_s ← R.x_izq
                12. Si dy > 0, o Si dy = 0 y P.v[i].y > R.y_sup, entonces
                        13. y_e ← R.y_inf
                        14. y_s ← R.y_sup
                15. Si no, entonces
                        16. y_e ← R.y_sup
                        17. y_s ← R.y_inf
                18. Si dx ≠ 0, entonces
                        19. tX_s ← ( x_s - P.v[i].x ) / dx
                20. Si no, compruebe Si R.x_izq ≤ P.v[i].x ≤ R.x_der, entonces
                        21. tX_s ← ∞
                22. Si no, entonces
                        23. tX_s ← -∞
```

```
                24. Si dy ≠ 0, entonces
                        25. tY_s ← ( y_s - P.v[i].y ) / dy
                26. Si no, compruebe Si R.y_inf ≤ P.v[i].y ≤ R.y_sup, entonces
                        27. tY_s ← ∞
                28. Si no, entonces
                        29. tY_s ← -∞
                // Ordenamos los dos puntos salientes
                30. Si tX_s < tY_s, entonces
                        31. t1_s ← tX_s
                        32. t2_s ← tY_s
                33. Si no, entonces
                        34. t1_s ← tY_s
                        35. t2_s ← tX_s
                36. Si t2_s > 0, entonces  // Calculamos t2_e
                        37. Si dx ≠ 0, entonces
                                38. tX_e ← ( x_e - P.v[i].x ) / dx
                        39. Si no, entonces
                                40. tX_e ← -∞
                        41. Si dy ≠ 0, entonces
                                42. tY_e ← ( y_e - P.v[i].y ) / dy
                        43. Si no, entonces
                                44. tY_e ← -∞
                        45. Si tX_e < tY_e, entonces
                                46. t2_e ← tY_e
                        47. Si no, entonces
                                48. t2_e ← tX_e
                        49. Si t1_s < t2_e, entonces  // El segmento no es visible
                                50. Si 0 < t1_s ≤ 1, entonces  // Agregamos un vértice de
giro
                                        51. Si tX_e < tY_e, entonces
                                                52. Resultado.Agregar( x_s, y_e )
                                        53. Si no, entonces
                                                54. Resultado.Agregar( x_e, y_s )

                                55. Si no, entonces
                                        56. Si 0 < t1_s y t2_e ≤ 1, entonces
                                                57. Si 0 < t2_e, entonces  // Agregamos una
intersección
                                                        58. Si tX_e > tY_e, entonces
                                                                59. Resultado.Agregar( x_e,
P.v[i].y + tX_e * dy )
                                                        60. Si no, entonces
                                                                61. Resultado.Agregar( P.v[i].x
+ tY_e * dx, y_e )
                                                62. Si no, entonces  // Agregamos el vértice
inicial
                                                        63. Resultado.Agregar( P.v[i].x,
P.v[i].y )
                                                64. Si t1_s < 1, entonces  // Agregamos una
intersección
                                                        65. Si tX_s < tY_s, entonces
                                                                66. Resultado.Agregar( x_s,
P.v[i].y + tX_s * dy )
                                                        67. Si no, entonces
                                                                68. Resultado.Agregar( P.v[i].x
+ tY_s * dx, y_s )

                                                69. Si no, entonces  // Agregamos el vértice
final
                                                        70. Resultado.Agregar( P.v[i+1].x,
P.v[i+1].y )
```

```
                    71. Si 0 < t2_s ≤ 1, entonces  // Agregamos un vértice de giro
                        72. Resultado.Agregar( x_s, y_s )

    73. Terminar( Resultado )
```

Agregar() se refiere a la operación básica para agregar un punto al final de la lista de vértices de un polígono.

Algoritmo de Weiler-Atherton

Este algoritmo se ideó originalmente para determinar la visibilidad de polígonos. Sin embargo, también sirve para determinar la intersección de dos polígonos, lo que implica que este mismo algoritmo se puede aplicar para recortar un polígono en un polígono de recorte. La ventaja de este algoritmo, comparado con otros, es que sirve para recortar polígonos convexos o cóncavos en cualquier polígono de recorte que también puede ser convexo o cóncavo; incluso acepta polígonos con agujeros. El resultado de este algoritmo es uno o varios polígonos que representan las partes visibles e interiores al polígono de recorte. Se supone, para cada polígono, una lista de vértices ordenada en el sentido de las agujas del reloj para representar el exterior del polígono. Una ordenación en el sentido contrario de las agujas del reloj sirve para describir cualesquier agujeros que contenga el polígono.

La estrategia se basa en recorrer los vértices y aristas del polígono a recortar que sean interiores al polígono de recorte. Si la arista está a punto de salirse del polígono de recorte, entonces cambiamos el recorrido al polígono de recorte, para continuar con su lista de vértices y aristas. Esto sugiere que debemos determinar todos los puntos de intersección entre ambos polígonos, teniendo en cuenta cuáles son entrantes y cuáles son salientes. Aplicamos las siguientes reglas al llegar a un punto de intersección en nuestro recorrido:

- Entrante: Recorra el polígono a recortar.
- Saliente: Recorra el polígono de recorte.

Ambos recorridos se hacen en el sentido de las agujas del reloj.

Por lo tanto, el algoritmo se divide en dos partes principales consecutivas: el cálculo de todas las intersecciones y posteriormente el recorrido de todos los puntos, según las reglas descritas anteriormente, para generar los polígonos resultantes.

Para formar cada polígono recortado, comenzamos con un punto de intersección entrante, agregando los demás puntos recorridos según las reglas establecidas hasta reencontrar ese mismo punto de intersección del comienzo, así cerrando el polígono.

Como ocurre en el recorte, es posible que las aristas de ambos polígonos no se crucen y por tanto no existan puntos de intersección. Si esto ocurriere, tenemos tres posibles casos a considerar según el polígono a recortar, *P*, y el polígono de recorte, *R*:

- *P* es interior a *R*, por lo que el resultado es *P*.
- *R* es interior a *P* o que es lo mismo, *P* engloba *R*, por lo que el resultado es *R*.
- *P* y *R* son polígonos que no se intersectan entre sí, ni en las aristas ni en sus interiores y por tanto no existe ningún polígono recortado resultante.

Ejemplo

Descripción

Retomamos el mismo ejemplo del método de Sutherland-Hodgman y Liang-Barsky. Tenemos un polígono de recorte, *R*, cuya lista de vértices es:

```
R : { (-1,3), (3,3), (3,-3), (-1,-3) }
```

Queremos uno varios polígonos, en tal polígono de recorte, a partir del polígono, *P*, descrito por la siguiente lista de vértices:

```
P : { (-2,1), (1,4), (4,3), (3,0), (0,-4), (-2,-4), (-3,-1) }
```

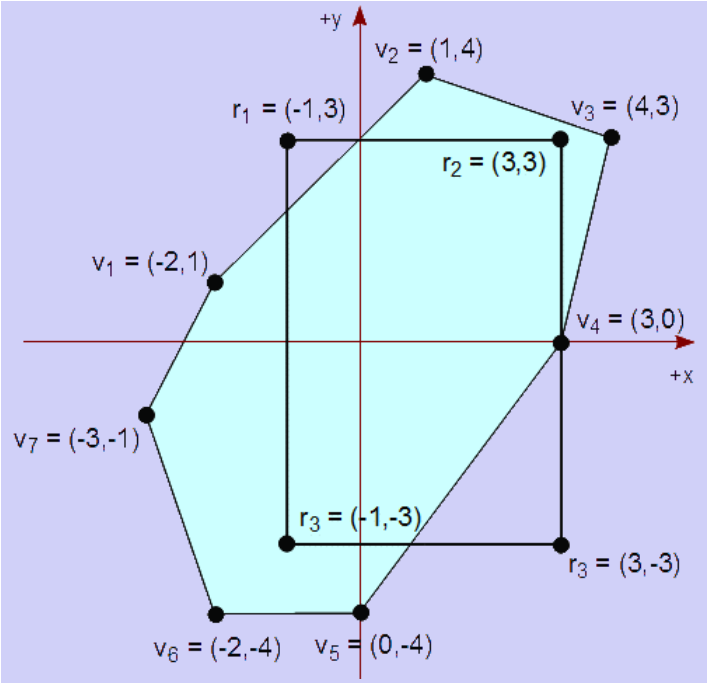


Figura 61 - Ejemplo del Algoritmo de Weiler-Atherton

Solución

Creamos una lista de polígonos, *Q*, que cada uno incluye una lista de vértices para representar los polígonos resultantes del recorte, inicialmente vacía:

```
Q1 : {}
```

Antes de empezar el recorte de nuestro polígono, calculamos todos los puntos de intersección. Como tenemos que insertar estos puntos correctamente en las dos listas de vértices, *P* y *R*, para que ambas listas sigan el sentido de las agujas del reloj. Para realizar esta ordenación, calculamos y guardamos sus parámetros del segmento, *t*. Adicionalmente, necesitamos saber cuáles de estos puntos de intersección son entrantes (en rojo en la figura 62) y cuáles son salientes (en morado en la figura 62). Los puntos son:

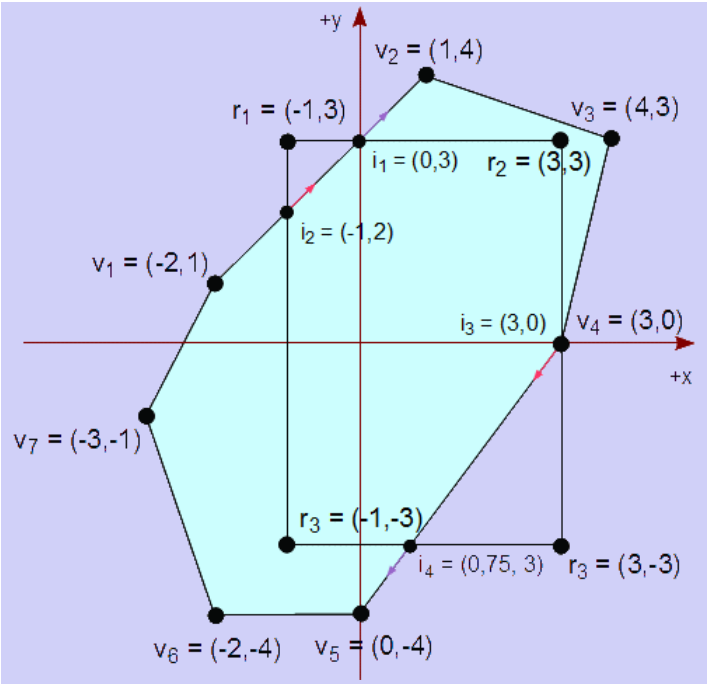


Figura 62 - Calculamos todas las intersecciones

```
i1 = (  0,   3 )  ⇐  t = 0,6666  (saliente)
i2 = ( -1,   2 )  ⇐  t = 0,3333  (entrante)
i3 = (  3,   0 )  ⇐  t = 0       (entrante)
i4 = (0,75,  3 )  ⇐  t = 0,75    (saliente)
```

Insertamos estos puntos de intersección correctamente en cada lista de vértices. i_1 e i_2 deben insertarse correctamente entre los vértices P.v_1 y P.v_2. Comparando los parámetros de i_1 e i_2, determinamos que el orden correcto es:

```
  v1        i2       i1       v2
(-2,1), (-1,2), (0,3), (1,4)
 t=0     t=0,33   t=0,66   t=1
```

El resultando de todas las inserciones correctas en las listas de vértices de ambos polígonos es:

```
P : { (-2,1), (-1,2), (0,3), (1,4), (4,3), (3,0), (3,0), (0,75, 3), (0,-4), (-2,-4),
(-3,-1) }

R : { (-1,3), (0,3), (3,3), (3,0), (3,-3), (0,75, 3), (-1,-3), (-1,2) }
```

Seguimos el recorrido, pero llegamos al punto i_1 que es saliente. Según las reglas, debemos cambiar de lista de vértices para continuar el recorrido en *R*. Agregando este punto i_1, por el momento, Q_1 contiene la siguiente lista:

```
Q1 : { (-1,2), (0,3) }
```

Ahora, seguimos con el recorrido del polígono, *R*. Para ello, buscamos el punto de intersección saliente, i_1, en la lista de *R*. Recorriendo *R*, nos encontramos con el vértice, R.r_2, el cual agregamos a Q_1. El siguiente punto es una intersección entrante, i_3, por lo que debemos cambiar a la lista de vértices del polígono, *P*. Ahora la lista de vértices, Q_1, es:

```
Q1 : { (-1,2), (0,3), (3,3), (3,0) }
```

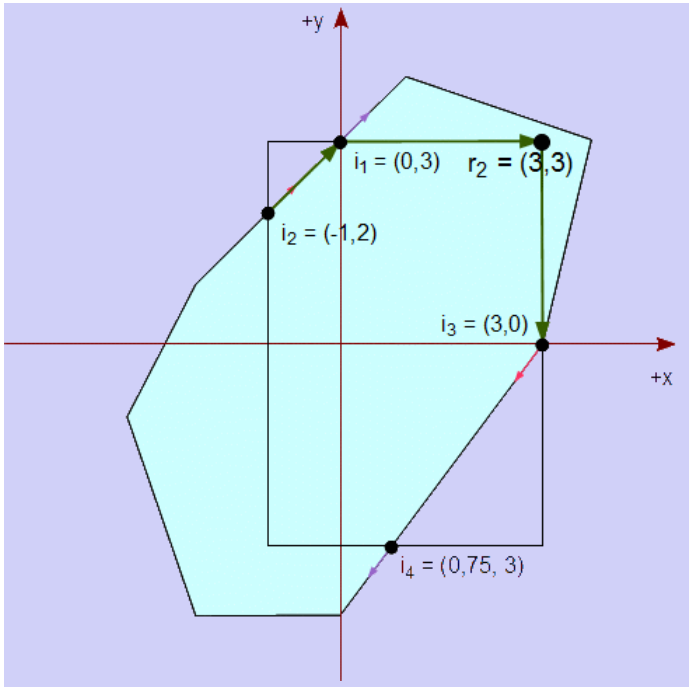


Figura 64 - Seguimos con el recorrido de R

Al saltar al polígono, *P*, buscamos el punto de intersección, i_3. Siguiendo la lista de *P*, a partir de i_3, nos encontramos con el punto de

intersección saliente, i_4, el cual agregamos a Q_1. Como se trata de un punto de intersección saliente, debemos cambiar a la lista de vértices del polígono, *R*. Por ahora la lista de vértices, Q_1, es:

```
Q1 : { (-1,2), (0,3), (3,3), (3,0),
(0,75, 3 ) }
```

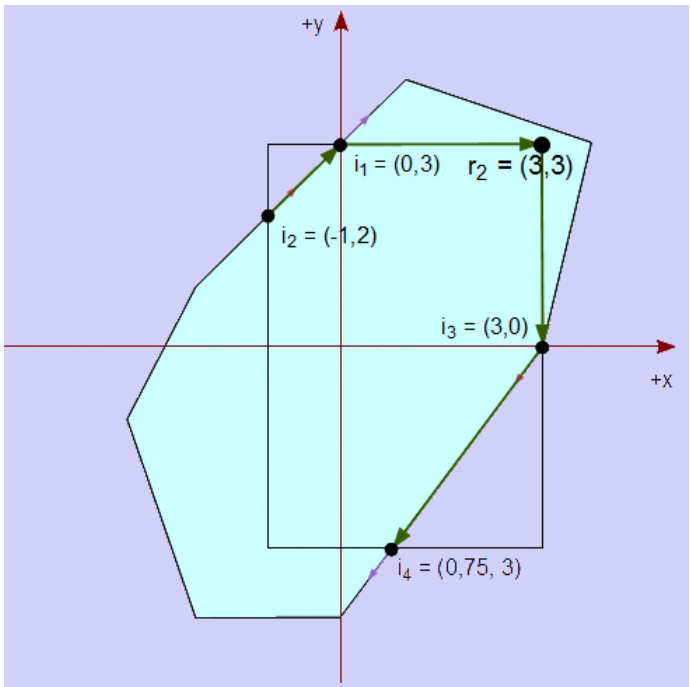


Figura 65 - Seguimos con el recorrido de P

Con la lista de vértices, *R*, buscamos el punto de intersección, i_4. Siguiendo la lista de *R*, a partir de i_4, nos encontramos con el vértice original, $R.r_3$, el cual agregamos a Q_1. El siguiente punto en *R* es de intersección entrante, i_1, que también se debería agregar a Q_1, pero ya lo visitamos previamente; y por tanto, se agregó en una etapa anterior del algoritmo. Como se trata de un punto de intersección entrante, debemos cambiar a la lista de vértices del polígono, *P*. Terminamos la lista de vértices, Q_1, que queda así:

```
Q1 : { (-1,2), (0,3), (3,3), (3,0),
(0,75, 3 ), (-1,-3) }
```

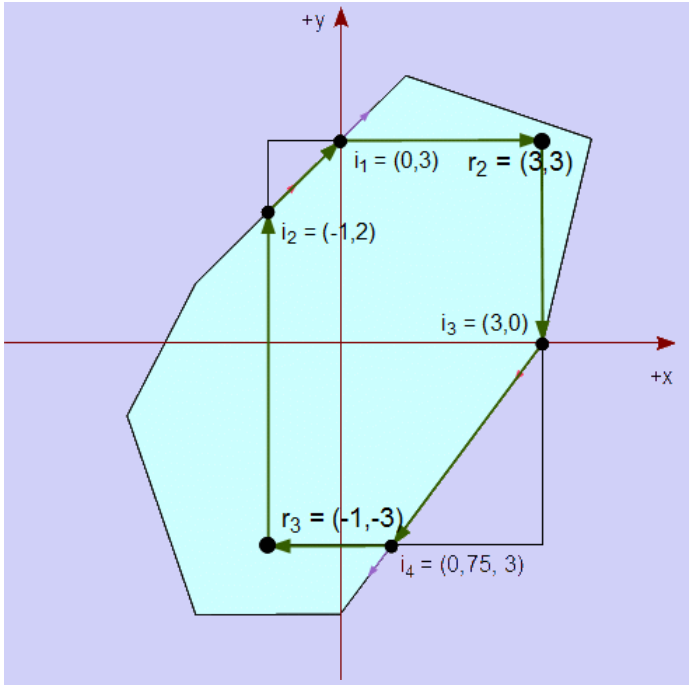


Figura 66 - Seguimos con el recorrido de R

Al terminar un polígono recortado, volvemos a comenzar el algoritmo con la lista de vértices, *P*. Buscamos el siguiente punto de intersección entrante que no hayamos visitado, ni por lo tanto hayamos agregado al polígono de

recorte, previamente. Como todos los puntos entrantes han sido visitados y usados anteriormente, finalizamos el recorte, dando lugar al polígono resultante, *Q*, que queda así:

```
Q : { (-1,2), (0,3), (3,3), (3,0),
(0,75, 3 ), (-1,-3) }
```

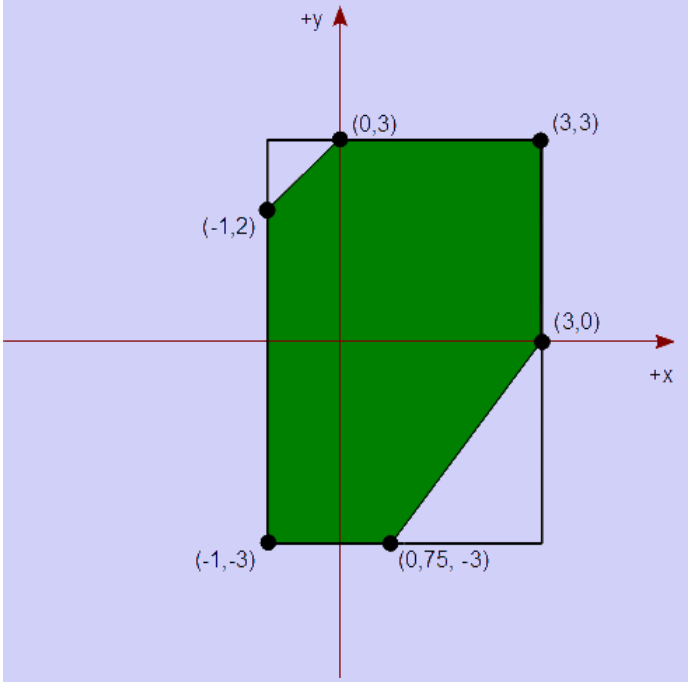


Figura 67 - Polígono recortado, Q

Algoritmo

El algoritmo hace uso de cierta información que agruparemos en la siguiente estructura especial, definida así:

```
PuntoInfo
{
  real t;
  Punto vértice;
  booleano entrante;
}
```

Esta información se asociará a los puntos de intersección que calcularemos. Como el algoritmo debe distinguir entre un punto original de un polígono dado y un punto de intersección calculado, agregaremos una propiedad a *Punto* llamada *original*; si es *verdadero*, entonces se trata de un punto original, y *falso* indicará que es una intersección. También agregamos otra propiedad a *Punto* llamada *visitado*; si es *verdadero*, entonces hemos procesado este punto en el algoritmo de la intersección, y *falso* indicará que aún está por procesar.

Presentamos el algoritmo para *Recortar()* que acepta cualquier tipo de polígono convexo o cóncavo cerrado:

```
ListaPolígonos Recortar( Polígono P, Polígono R )
        1. Crear ListaPolígonos: Resultado ← vacía
        2. P.Agregar( P.v[1] )  // Agregamos el primer vértice al final
        3. R.Agregar( R.v[1] )
        4. ExistenIntersecciones ← Calcular_Intersecciones( P, R )
        5. Si ExistenIntersecciones = verdadero, entonces
                6. Resultado ← Intersectar( P, R, Resultado )
```

```
        7. Si no, entonces
                8. ContienePunto ← Acepta( P, R.v[1] )
                9. Si ContienePunto = verdadero, entonces
                        10. Resultado ← R
                11. Si no, entonces
                        12. ContienePunto ← Acepta( R, P.v[1] )
                        13. Si ContienePunto = verdadero, entonces
                                14. Resultado ← P
                        15. Si no, entonces
                                16. Resultado ← vacía
        17. Terminar( Resultado )
```

Exponemos el algoritmo de *Calcular_Intersecciones()* para calcular los puntos de intersección de las aristas entre los dos polígonos. Simultáneamente, agregamos estos puntos de intersección a los dos polígonos. Adicionalmente indicamos cuáles puntos son entrantes y cuáles son salientes.

```
booleano Calcular_Intersecciones( ref Polígono P, ref Polígono R )
        1. ExistenIntersecciones ← falso
        2. A ← P.v1
        3. B ← Siguiente_Original( P, A )
        4. Mientras que, B ≠ P.v1
                5. C ← R.v1
                6. D ← Siguiente_Original( R, C )
                7. Mientras que, D ≠ R.v1, repetir
                        8. tp ← Calcular_Parámetro( A, B, C, D )
                        9. Si 0 ≤ tp ≤ 1, entonces
                                10. tr ← Calcular_Parámetro( C, D, A, B )
                                11. Si 0 ≤ tr ≤ 1, entonces
                                        12. ExistenIntersecciones ← verdadero
                                        13. nuevo.vértice ← A + tp*(B-A)
                                        14. nuevo.vértice.original ← falso
                                        15. nuevo.t ← tp
                                        16. nuevo.entrante ← Es_Entrante( A, B, C, D )
                                        17. Insertar( P, A, B, nuevo )
                                        18. nuevo.t ← tr
                                        19. Insertar( Q, C, D, nuevo )
                        20. C ← D
                        21. D ← Siguiente_Original( R, C )
                22. A ← B
                23. B ← Siguiente_Original( P, A )
        24. Terminar( ExistenIntersecciones )
```

El algoritmo de *Siguiente_Original()* requiere un polígono y un punto que pertenece a tal polígono. Recorrerá todos los puntos en busca del siguiente que es un vértice - un punto original del polígono.

```
Punto Siguiente_Original( Polígono P, Punto A )
        1. Resultado ← P.Siguiente( A )
        2. Mientras que, Resultado.original = falso, repetir
                3. Resultado ← P.Siguiente( Resultado )
        4. Terminar( Resultado )
```

Hacemos uso del algoritmo básico *Siguiente()* que obtendrá el siguiente vértice en la lista que forma el polígono.

El algoritmo para *Calcular_Parámetro()* requiere los puntos extremos de cada segmento para calcular el valor del parámetro del primer segmento del punto de intersección de ambos segmentos.

```
real Calcular_Parámetro( Punto P0, Punto P1, Punto Q0, Punto Q1 )
        1. N ← ( Q0y - Q1y, Q1x - Q0x )
        2. numerador ← N·(P0 - Q0)
        3. denominador ← -N·(P1 - P0)
        4. Si denominador ≠ 0, entonces
                5. t ← numerador / denominador
        6. Si no, entonces
                7. t ← ∞
        8. Terminar( t )
```

He aquí el algoritmo de *Es_Entrante()*, que determina si el segmento, descrito por el vector **U**, es entrante (verdadero) al cruzar la arista descrita por el vector **V**. Esto implica que su punto de intersección, tambié es entrante.

```
booleano Es_Entrante( Vector U, Vector V )
        1. N ← ( -Vy, Vx )
        2. denominador ← N·U
        3. Si denominador < 0, entonces
                4. Terminar( verdadero )
        5. Si no, entonces
                6.Terminar( falso )
```

El siguiente algoritmo describe *Insertar()*, que acepta un polígono, dos puntos extremos de una arista, y el nuevo punto a insertar correctamente.

```
Insertar( ref Polígono P, Punto A, Punto B, PuntoInfo nuevo )
        1. Actual ← A
        2. Sig ← P.Siguiente( A )
        3. terminar ← falso
        4. Mientras que, terminar = falso  y  Sig.original = falso, repetir
                5. Si nuevo.t < Sig.t, entonces
                        6. P.Insertar_Nuevo( Actual, nuevo )
                        7. terminar ← verdadero
                8. Si no, entonces
                        9. Actual ← Sig
                        10. Sig ← P.Siguiente( Actual )
                11. Resultado ← P.Siguiente( Resultado )
        12. P.Insertar_Nuevo( Actual, nuevo )
        13. Terminar
```

El algoritmo básico del polígono, *Insertar_Nuevo()*, sirve para agregar un nuevo punto a ello, justo después del punto indicado.

Exponemos el algoritmo de *Intersectar()* para determinar los polígonos de intersección entre dos polígonos dados.

```
ListaPolígonos Intersectar( Polígono P, Polígono R )
        1. Crear ListaPolígonos: Actual ← { P, R }   // Actual[1] = P, Actual[2] = R
        2. Crear ListaPolígonos: Resultado ← vacía
        3. Crear Polígono: Auxiliar ← vacío
```

```
        4. índice ← Buscar_Punto_Entrante_No_Visitado( P )
        5. Mientras que, índice > 0, repetir
                6. n ← 1
                7. Mientras que, Actual[n].v[índice].visitado = falso, repetir
                        8. Actual[n].v[índice].visitado = verdadero
                        9. Auxiliar.Agregar( Actual[n].v[índice].vértice )
                        10. Si Actual[n].v[índice].original = falso, entonces
                                11. Si Actual[n].v[índice].entrante = falso, entonces
// Cambio de polígono
                                        12. n_ant ← n
                                        13. Si n = 1, entonces
                                                14. n ← 2
                                        15. Si no, entonces
                                                16. n ← 1
                                        17. índice ← Buscar_Punto( Actual[n],
Actual[n_ant].v[índice].vértice )
                        18. índice ← índice + 1
                        19. Si índice > Actual[n].Cantidad_Vértices(), entonces
                                20. índice ← 1
                21. Resultado.AgregarPolígono( Auxiliar )
                22. Auxiliar.Vaciar()
                23. índice ← Buscar_Punto_Entrante_No_Visitado( P )
        24. Terminar( Resultado )
```

He aquí el algoritmo de *Buscar_Punto_Entrante_No_Visitado()* para encontrar el punto entrante sin marcar como *visitado* del polígono dado. El algoritmo retornará el índice al punto encontrado o 0 (cero), para indicar que no existe ninguno.

```
entero Buscar_Punto_Entrante_No_Visitado( Polígono P )
        1. Para: índice ← 1 hasta P.Cantidad_Vértices(), repetir
                2. Si P.v[índice].original = falso  y  Si P.v[índice].entrante =
verdadero  y  Si P.v[índice].visitado = falso, entonces
                        3. Terminar( índice )
        4. Terminar( 0 )
```

Presentamos el algoritmo de *Buscar_Punto()* para encontrar el punto indicado en el polígono dado. El algoritmo retornará el índice al punto encontrado o 0 (cero), para indicar que no se encontró.

```
entero Buscar_Punto( Polígono P, Punto I )
        1. Para: índice ← 1 hasta P.Cantidad_Vértices(), repetir
                2. Si P.v[índice].vértice = I, entonces
                        3. Terminar( índice )
        4. Terminar( 0 )
```

Observaciones

Para aquellas implementaciones del algoritmo de Sutherland-Hodgman para hardware, podemos descomponer este algoritmo en varias etapas. Cada etapa manipula un vértice para cada arista y así no tenemos que usar una lista de vértices intermediaria. Esta mejora se puede conseguir con un procesamiento paralelo o con un solo procesador configurado por etapas. Si un vértice es validado al terminar todas las etapas de recorte, entonces se agrega a la lista final de vértices. De lo contrario, el vértice no continúa en el procesamiento.

Una mejora del algoritmo de Weiler-Atherton es mantener una lista de puntos entrantes, en lugar de buscarlos en el polígono a recortar. Dejamos su implementación como un ejercicio.

También podemos usar este algoritmo para conseguir la unión de dos polígonos. En lugar de tratar los puntos de intersección entrantes como el principio del polígono resultante y los salientes para alternar entre polígonos, invertimos esta lógica. Tratamos los puntos de intersección entrantes para alternar entre polígonos y los salientes como el principio del polígono resultante. También dejamos esta implementación como un ejercicio.

Concluyendo, el algoritmo de Nicholl-Lee-Nicholl es actualmente el mejor de los algoritmos de recorte de segmentos que hemos tratado. Los algoritmos de Liang-Barsky y Cohen-Sutherland siguen siendo populares, y en este último caso, el algoritmo es muy fácil de implementar. Estos tres algoritmos se basan en un rectángulo vertical de recorte. El algoritmo de Cyrus-Beck se basa en un polígono convexo de recorte, pero requiere varios cálculos costosos en comparación con los algoritmos anteriores. Para los algoritmos de recorte de polígonos, los algoritmos presentados son fáciles de entender y de describir, pero no son recomendados porque requieren varios cálculos, y además existen otros algoritmos de mejor rendimiento.

Volviendo a nuestro esquema de las etapas del procesamiento de datos geométricos a colores asignados a cada píxel de la pantalla o puntos de impresión en papel, agregamos el proceso de recorte como una nueva etapa. Nuestro esquema es ahora el siguiente:

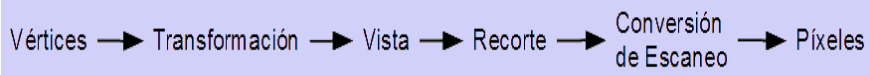


Figura 68 - Procesamiento Geométrico

Ejercicios

Enlace al paquete de este capítulo.

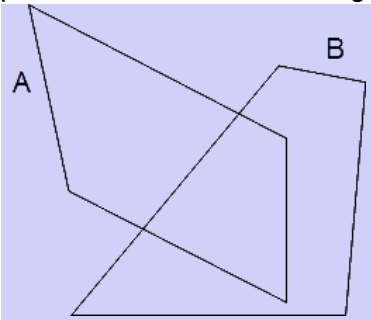
1. Como mencionamos en la explicación del algoritmo de Cohen-Sutherland, podemos mejorarlo un poco para que no recalcule la pendiente de la intersección en cada iteración.
   a. Implemente el algoritmo original,
   b. Implemente otra versión del algoritmo con la mejora mencionada, y
   c. Escriba un programa para comparar el rendimiento de ambos algoritmos. Para el programa, cree aleatoriamente varios segmentos - decenas de millares - para pasarlos por los dos algoritmos de a) y b) para ser recortados. Cronometre los procesamientos de ambos algoritmos y muestre los resultados de la comparativa.
2. Optimice el algoritmo de Nicholl-Lee-Nicholl para reusar multiplicaciones y restas y así no hay que invocar a *Izquierda()*, ni a *Derecha()*, ni tampoco a *Intersección_Vertical()* ni a *Intersección_Horizontal()*. Por ejemplo, en el algoritmo de *Recortar_Interior()*, podemos sustituir el comienzo con los siguientes pasos, para no tener que usar *Izquierda()*:

```
2. QPX ← Q.x - P.x
3. QPY ← Q.y - P.y
4. A ← R.x_izq - P.x
5. B ← R.y_sup - P.y
6. Si A*QPX + B*QPY > 0, entonces
```

3. Implemente las mejoras mencionadas anteriormente para el algoritmo Weiler-Atherton. Cree otra lista de puntos, o referencias a los puntos, para guardar las intersecciones entrantes, que previamente hemos calculado. Esta lista contendrá menos puntos que la lista de los vértices originales y puntos de intersección. Ahora, sólo tendremos que consultar esta

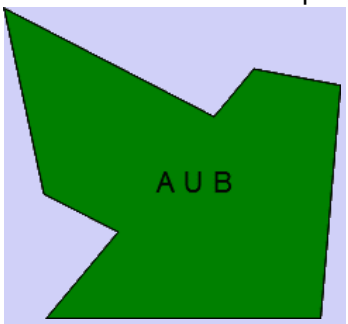
lista, en lugar de buscar las intersecciones entre todos los puntos, para conseguir aquellas intersecciones entrantes aún sin visitar.

4. Como mencionamos anteriormente, podemos usar el algoritmo de Weiler-Atherton para el modelado. Con las operaciones básicas de conjuntos - unión, intersección, y diferencia - podemos construir nuevas figuras geométricas a partir de dos figuras básicas, como mínimo.



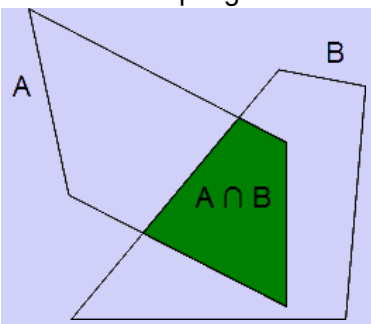
Ejercicio 4 - Polígonos A y B

- A ∪ B. Para la unión, buscamos cualesquier intersecciones entre los dos polígonos A y B. Comenzamos con el recorrido de A, el polígono a recortar, o de B, el polígono de recorte. Cambiamos de polígono al encontrarnos con intersecciones.



Ejercicio 4 - Unión de A y B

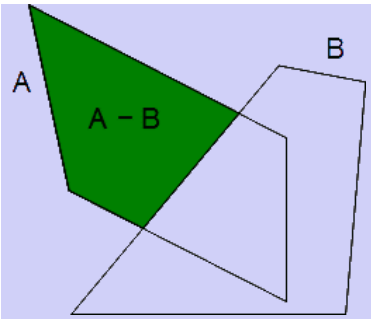
- A ∩ B. La intersección es justamente el algoritmo original de recorte de Weiler-Atherton. Recorremos el polígono A al encontrarnos con las intersecciones entrantes y recorremos el polígono B al encontrarnos con las intersecciones salientes.



Ejercicio 4 - Intersección de A y B

- A − B. Para la diferencia, comenzamos con el polígono A y eliminamos partes de ello con el polígono B. Usando el algoritmo de Weiler-Atherton, recorremos el polígono A al

encontrarnos con las intersecciones salientes y recorremos el polígono B al encontrarnos con las intersecciones entrantes.



Ejercicio 4 - Diferencia de A y B

Cree un programa que permita al usuario generar polígonos con estas operaciones.

Apéndice 1: Geometría

Daremos un repaso general de la geometría, haciendo hincapié en la trigonometría.

Definiciones

Triángulo: Polígono cerrado de tres ángulos, y por tanto con tres lados.

- Rectángulo: Triángulo que tiene un ángulo recto; o sea, de 90°. Cada lado que forma el ángulo de 90° se llama *cateto* y el lado, que une los catetos, se llama *hipotenusa*. La hipotenusa es siempre mayor que cualquiera de los dos catetos.
- Isósceles: Triángulo que tiene dos lados de igual longitud y por tanto, dos de los ángulos son iguales.
- Equilátero: Triángulo que tiene los tres lados de igual longitud al igual que sus tres ángulos: 60°.

Cuadrilátero: Polígono cerrado de cuatro lados.

- Paralelogramo: Un cuadrilátero cuyos lados opuestos son paralelos y de igual longitud.
- Rombo: Un paralelogramo cuyos lados son de igual longitud.
- Rectángulo: Un paralelogramo cuyos lados forman ángulos rectos (90°).
- Cuadrado: Un rectángulo cuyos lados miden la misma longitud.

Geometría General

Líneas Rectas

Ecuaciones

Ecuaciones para una línea recta:

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

Esta ecuación describe una línea recta que pasa por el punto (x_0, y_0), con una pendiente m. Resolviendo para y, obtenemos la siguiente ecuación:

$$y = m\ x + b,$$

En esta ecuación, se describe una línea recta, con una pendiente m, que corta el eje Y (la ordenada) en el punto $(0, b)$, y corta el eje X (la abscisa) en el punto $(-b/m, 0)$.

Si x es una constante, x_c, entonces nuestra ecuación describe una línea recta vertical:

$$x = x_c,$$

Si y es una constante, y_c, entonces nuestra ecuación describe una línea recta horizontal:

$$y = y_c$$

Propiedades entre dos líneas rectas

Dos líneas rectas son *coincidentes*, cuando una línea sobrepone a la otra. Dicho de otro modo, todos los puntos de ambas líneas son comunes entre sí.

Dos líneas rectas son *paralelas*, cuando una línea no cruza la otra. Dicho de otro modo, no existe un punto común para ambas líneas.

Dos líneas rectas son *perpendiculares*, cuando una línea corta la otra formando un ángulo recto (90°).

Dos líneas rectas son *oblicuas*, cuando las líneas que intersectan no son perpendiculares.

Trigonometría

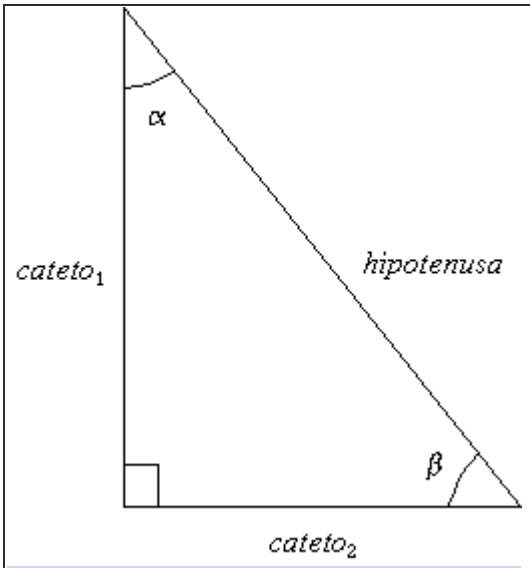
Para un triángulo rectángulo, se establecen las siguientes fórmulas:

$$\text{hipotenusa}^2 = (\text{cateto}_1)^2 + (\text{cateto}_2)^2 \quad \text{(Teorema de Pitágoras)}$$

$$\text{sen } x = \frac{\text{cateto opuesto}}{\text{hipotenusa}}$$

$$\cos x = \frac{\text{cateto contiguo}}{\text{hipotenusa}}$$

$$\text{tg } x = \frac{\text{cateto opuesto}}{\text{cateto contiguo}}$$

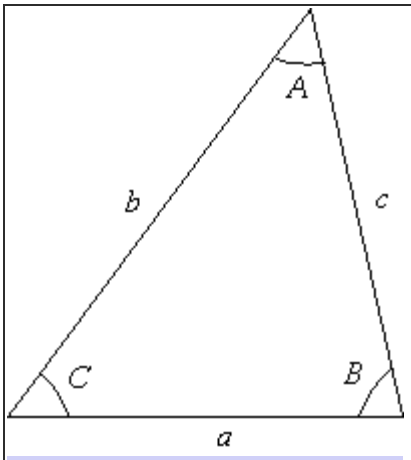


Triángulo rectángulo

Sean *a*, *b*, y *c* las longitudes de cada lado de un triángulo arbitrario y sean *A*, *B*, y *C* los ángulos opuestos a los lados mencionados respectivamente, se establecen las siguientes fórmulas:

$$c^2 = a^2 + b^2 - 2ab\cos C \quad \text{(Ley de cosenos)}$$

$$\frac{\text{sen } A}{a} = \frac{\text{sen } B}{b} = \frac{\text{sen } C}{c} \quad \text{(Ley de senos)}$$



Triángulo arbitrario

Fórmulas generales - Identidades:

$\text{sen}(-x) = -\text{sen } x$	$\cos(-x) = \cos x$	

		$$tg\ x = \frac{sen\ x}{cos\ x}$$
$$cosec\ x = \frac{1}{sen\ x}$$	$$sec\ x = \frac{1}{cos\ x}$$	$$cotg\ x = \frac{cos\ x}{sen\ x} = \frac{1}{tg\ x}$$
$sen^2\ x + cos^2\ x = 1$	$tg^2\ x + 1 = sec^2\ x$	$1 + cotg^2\ x = cosec^2\ x$
$sen(x+y) = sen(x)\ cos(y) + cos(x)\ sen(y)$	$cos(x+y) = cos(x)\ cos(y) - sen(x)\ sen(y)$	$$tg(x+y) = \frac{tg\ x + tg\ y}{1 - tg(x)\ tg(y)}$$
$sen\ 2x = 2\ sen(x)\ cos(x)$	$cos\ 2x = cos^2\ x - sen^2\ x$	
$sen^2(x/2) = (1 - cos\ x)\ /\ 2$	$cos^2(x/2) = (1 + cos\ x)\ /\ 2$	

Apéndice 2: Vectores

Los vectores sirven para agrupar valores numéricos ordenados linealmente. Los vectores tienen ciertas propiedades matemáticas que pueden resultar en cálculos más rápidos. Para la teoría de gráficos, usamos vectores en la representación de ciertos objetos y conceptos descritos por su orientación - vector - y un punto - localización. Generalmente, usaremos vectores de 4 elementos, para describir la orientación de ejes de un sistema de coordenadas o incluso la orientación de la cámara de una escena, por ejemplo.

Definiciones

Un **vector** es un conjunto de elementos que geométricamente describe un segmento orientado. Este segmento conlleva suficiente información para conocer la orientación, dirección, y su magnitud (o longitud). Visto de otra manera:

$$\mathbf{v} = (v_1, v_2, v_3, v_4, \ldots, v_n)$$

Aquí **v** es un vector de dimensión (u orden) n, ya que tiene *n* elementos o **componentes**.

Para obtener un vector, requerimos dos puntos, ya que un vector es al fin y al cabo un segmento. Las coordenadas de ambos puntos se restan para calcular el vector. Esto es, tenemos los dos siguientes puntos:

$$A = (a_1, a_2, a_3, a_4, \ldots, a_n)$$
$$B = (b_1, b_2, b_3, b_4, \ldots, b_n)$$

Para conseguir un vector desde el punto *A* al punto *B*, realizamos la siguiente operación:

v = B-A = (b_1-a_1, b_2-a_2, b_3-a_3, b_4-a_4, …, b_n-a_n)

Podemos ver el resultado gráficamente en la *Figura 1*:

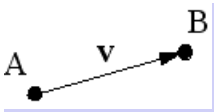
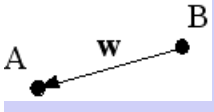


Figura 1 - Vector

Orientación

También obtendremos un vector desde *B* hasta a *A*. Sin embargo, la orientación o el sentido del vector **w** es contraria a la del vector **v**. El vector se obtendría de esta manera:

$$\mathbf{w} = A-B = (a_1-b_1, a_2-b_2, a_3-b_3, a_4-b_4, \ldots, a_n-b_n)$$



Podemos comparar gráficamente la diferencia de cada vector viendo la *Figura 2*:

Figura 2 - Vector

Por lo tanto, tenemos que,

$$\mathbf{w} = -\mathbf{v}$$

El signo negativo para vectores significa que el vector tiene una orientación o sentido contrario.

Pendiente

Un vector tiene dirección que es lo mismo que la pendiente. Como un vector contiene la diferencia entre las coordenadas de dos puntos, entonces podemos calcular fácilmente la pendiente. Usamos la siguiente fórmula para vectores en 2D:

$$\mathbf{v} = (v_1, v_2)$$
$$m = v_2 / v_2$$

Módulo

Un vector también tiene longitud o magnitud, llamado **módulo** del vector. ésta se calcula de la siguiente manera usando el teorema de Pitágoras:

$$\mathbf{v} = (v_1, v_2, v_3, v_4, \ldots, v_n)$$
$$\|\mathbf{v}\|^2 = (v_1)^2 + (v_2)^2 + (v_3)^2 + (v_4)^2 + \ldots + (v_n)^2$$

Si un vector tiene como módulo 1, entonces se trata de un *vector unitario*.

Vector Nulo

Hay que hacer mención del vector cero, **0**, el cual se define así:

$$\mathbf{0} = (0, 0, 0, \ldots, 0)$$

Este vector simplemente no tiene ni dirección, ni por tanto orientación, pero tiene magnitud 0 (cero). Por estas razones, se le suele llamar el *vector nulo*.

Equivalencia

Dos vectores son iguales si sus componentes homólogos son iguales. Expresamos este concepto de esta manera:

$$\mathbf{u} = (u_1, u_2, u_3, u_4, \ldots, u_n)$$
$$\mathbf{v} = (v_1, v_2, v_3, v_4, \ldots, v_n)$$

$$\mathbf{u} = \mathbf{v}$$

si y sólo si,

$u_1 = v_1$

$u_2 = v_2$

$u_3 = v_3$

$u_4 = v_4$

$\vdots$

$u_n = v_n$

Los vectores iguales entre sí tienen la misma dirección, orientación, y magnitud. Esto implica que los vectores son paralelos entre sí. Esto se puede apreciar geométricamente en la *Figura 3*.

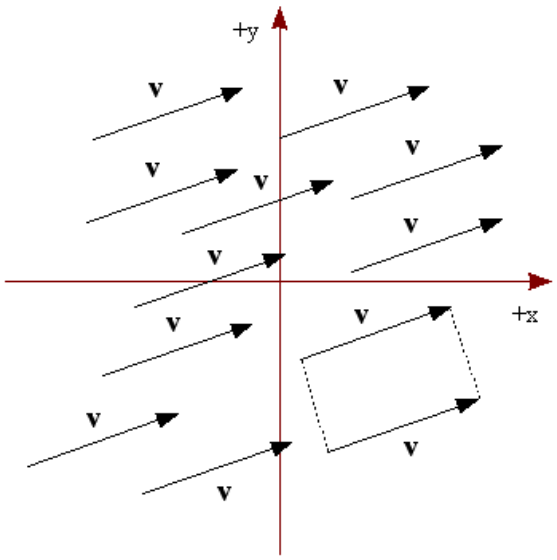


Figura 3 - Equivalencia

Ejemplos

Veamos algunos ejemplos.

Ejemplo 1

Calcule el vector **PQ**, su módulo, y la pendiente *m* a partir de estos dos puntos:

P = (-2, 8) y Q = (4, -4)

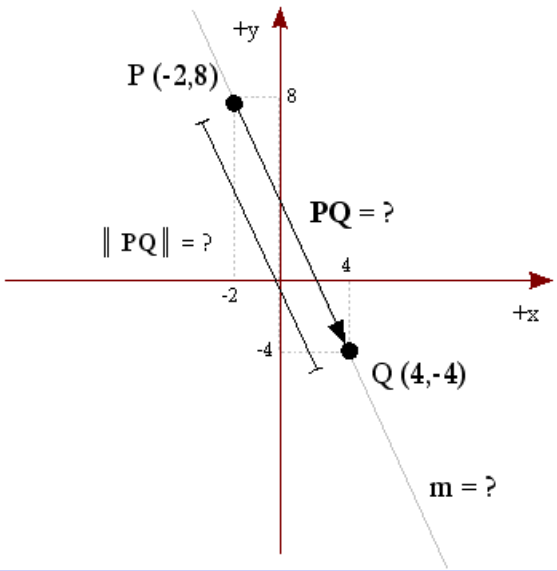


Figura 4 - Ejemplo 1 Vectores

Solución

Calculamos el vector **PQ**,

PQ = Q-P = (4, -4) - (-2, 8) = (4-(-2), -4-8) = (6, -12)

La pendiente *m* es:

$m = -12 / 6 = -2$

El módulo del vector **v** es:

$\|\mathbf{PQ}\| = \sqrt{(6^2 + (-12)^2)} = \sqrt{(36 + 144)} = \sqrt{180} \cong 13,4164$

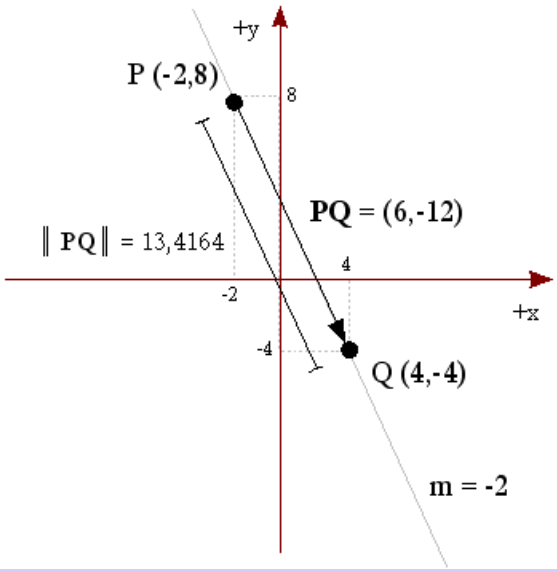


Figura 5 - Ejemplo 1 Solución

Ejemplo 2

Determine las coordenadas del punto **B**, dada la siguiente información:

- El punto A = (2, -1) y *B* pertenecen a la misma línea recta
- La pendiente de tal línea es m = 0,75
- El punto *B* se encuentra en el tercer cuadrante y a 5 unidades del punto *A*

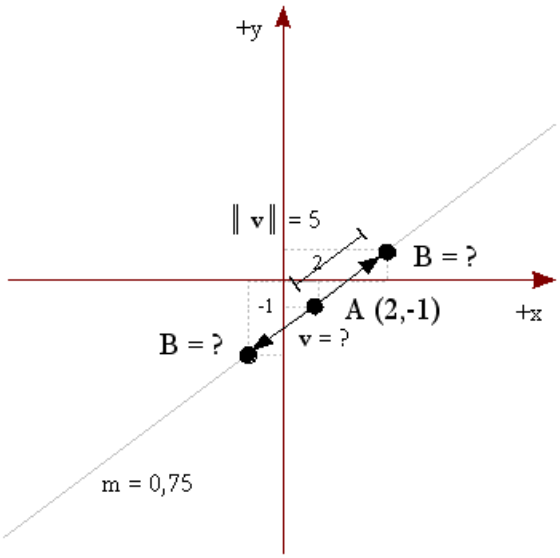


Figura 6 - Ejemplo 1 Vectores

Solución

Vamos a intentar calcular el vector **AB**, para ver hasta dónde podemos llegar y así ver qué información nos hace falta.

$\mathbf{v} = \mathbf{AB} = B-A = (x, y) - (2, -1) = (x-2, y+1)$

O sea, tenemos lo siguiente:

$$\mathbf{v} = (v_x, v_y) = (x-2, y+1),$$

esto implica que,

$$v_x = x-2$$
$$v_y = y+1$$

Obviamente, necesitamos determinar los valores de *x* e *y* que forman las coordenadas del punto *B*.

Como nos dan la pendiente *m*, usaremos su fórmula:

$$m = v_y / v_x \Rightarrow 0,75 = v_y / v_x \Rightarrow v_y = 0,75\ v_x$$

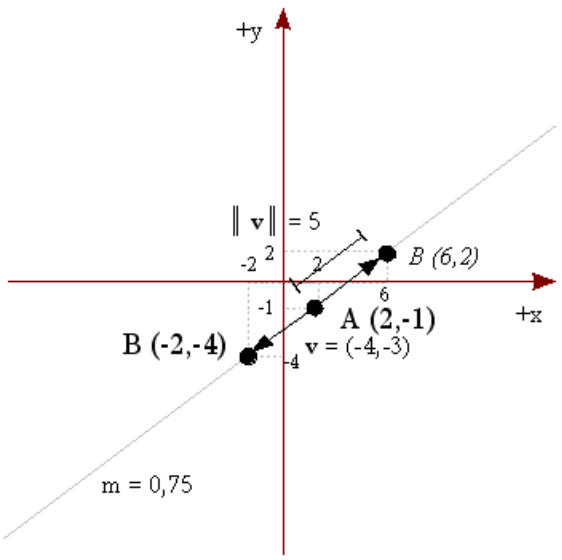


Figura 7 - Ejemplo 2 Solución

También nos dan la distancia entre ambos puntos: 5. La distancia es en realidad el módulo del vector **v**, la cual conocemos su fórmula:

$$\|\mathbf{v}\|^2 = (v_x)^2 + (v_y)^2$$

Con la información dada, tenemos que,

$$5^2 = (v_x)^2 + (v_y)^2$$

Sustituimos los componentes del vector **v** con la ecuación que obtuvimos - basada en la pendiente. Esto es,

$$25 = (v_x)^2 + (0,75\ v_x)^2 \Rightarrow 25 = (v_x)^2 + 0,5625\ (v_x)^2 \Rightarrow 25 = 1,5625\ (v_x)^2 \Rightarrow \sqrt{16} = v_x \Rightarrow v_x = 4$$

Volvemos a la ecuación de la pendiente para averiguar v_y,

$$v_y = 0,75\ v_x = 0,75*4 = 3$$

Ahora, sustituimos los componentes del vector **v** en las primeras ecuaciones que obtuvimos. Esto es,

$$v_x = x-6 \Rightarrow 4 = x-2 \Rightarrow x = 6$$
$$v_y = y-3 \Rightarrow 3 = y+1 \Rightarrow y = 2$$

Por lo tanto, el punto B = (6, 2). No obstante, se nos olvida el último dato importante: el punto *B* se encuentra en el tercer cuadrante. Revisando las coordenadas que hemos calculado, vemos que el punto (6, 2) pertenece al primer cuadrante y no al tercero. Recordamos, que un vector tiene orientación y por tanto es posible que estemos ante un vector con un sentido contrario al que pensábamos.

Esto es posible ya que calculamos la pendiente basándonos en que los componentes del vector eran ambos positivos, pero también puede darse el caso de que ambos sean negativos. Esto es,

$$m = -v_y / -v_x \Rightarrow 0,75 = -v_y / -v_x$$

Por consiguiente, tenemos que,

$$\mathbf{v} = (-4, -3)$$

Esto implica que,

$$v_x = x-2 \Rightarrow -4 = x-2 \Rightarrow x = -2$$
$$v_y = y+1 \Rightarrow -3 = y+1 \Rightarrow y = -4$$

Consecuentemente, nuestro punto B = (-2, -4) es el que se nos pide ya que yace en el tercer cuadrante.

Operaciones de Vectores

Existen cuatro operaciones de vectores básicas: multiplicación escalar-vector, suma vector-vector, resta vector-vector, producto escalar, y producto vectorial.

Multiplicación Escalar-Vector

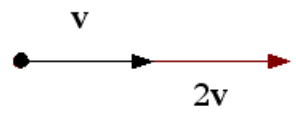
Es la multiplicación del escalar *s* por cada elemento del vector **v**.

$$s\mathbf{v} = s * (v_1, v_2, v_3, v_4, \ldots, v_n) = (sv_1, sv_2, sv_3, sv_4, \ldots, sv_n)$$

Este tipo de multiplicación tiene las siguientes propiedades:

- conmutativa: s**v** = **v**s
- asociativa: s(t**v**) = (st)**v**
- distributiva: (s+t)**v** = s**v** + t**v**

Como el escalar *s* modifica todos los elementos de un vector igualmente, entonces el vector en sí cambia proporcionalmente. Esto implica que la dirección no varía, pero posiblemente sí su orientación y magnitud. Veamos los casos:

Condición	**Efecto**	**Ejemplo**
Si *s* > *1*	el módulo de **v** aumenta	 Figura 8 - s>1
Si *s* = *1*	**v** no varía y por tanto, su módulo tampoco	

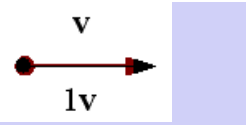


Figura 9 - s=1

Si *0 < s <* *1* el módulo de **v** disminuye

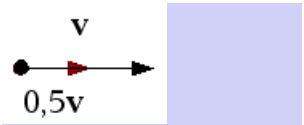


Figura 10 - 0<s<1

Si *s = 0* **v** = **0** y por tanto, su módulo es 0

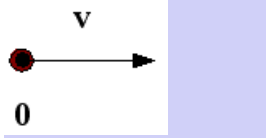


Figura 11 - s=0

Si *s < 0* el módulo de **v** varía según la cantidad de *s*, pero su signo hace invertir la orientación o sentido del vector **v**

Figura 12 - s<0

Suma Vector-Vector

Se suman los elementos correspondientes de los dos vectores. La suma sólo se puede hacer si los dos vectores son del mismo orden o dimensión: el número de elementos es el mismo para el vector **u** y **v**:

$$\mathbf{u} + \mathbf{v} = (u_1, u_2, u_3, u_4, \ldots, u_n) + (v_1, v_2, v_3, v_4, \ldots, v_n) = (u_1+v_1, u_2+v_2, u_3+v_3, u_4+v_4, \ldots, u_n+v_n)$$

La suma de vectores acepta las propiedades:

- conmutativa: **u** + **v** = **v** + **u**
- asociativa: **u** + (**v** + **w**) = (**u** + **v**) + **w**
- distributiva (con escalares): s(**u** + **v**) = s**u** + s**v**

Por supuesto, todos los vectores han de ser de orden *n*.

Geométricamente, el resultado de la suma de vectores es otro vector llamado el *vector resultante*. Como los vectores no están fijados a un punto, podemos "mudarlos" a nuestro antojo. Si colocamos uno de los vectores para poder encadenarlo a nuestro otro vector sumando, observamos que el resultante forma un triángulo junto con los otros dos sumandos. La forma de encadenar los vectores es colocando la cola de un vector sobre la cabeza de otro. Esto se puede comprobar en las *Figura 13* y *Figura 14*.

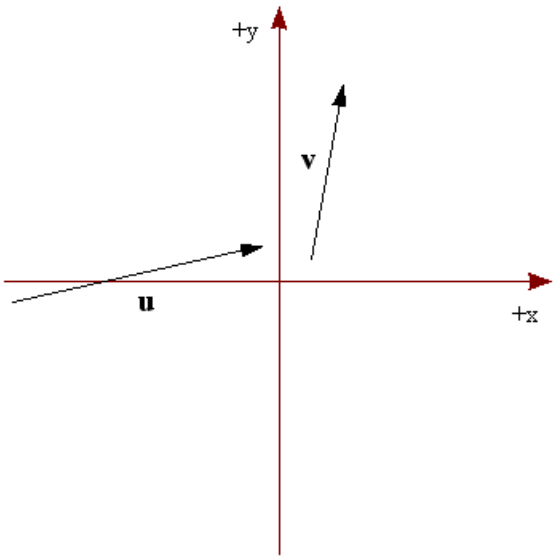


Figura 13 - Vectores u y v

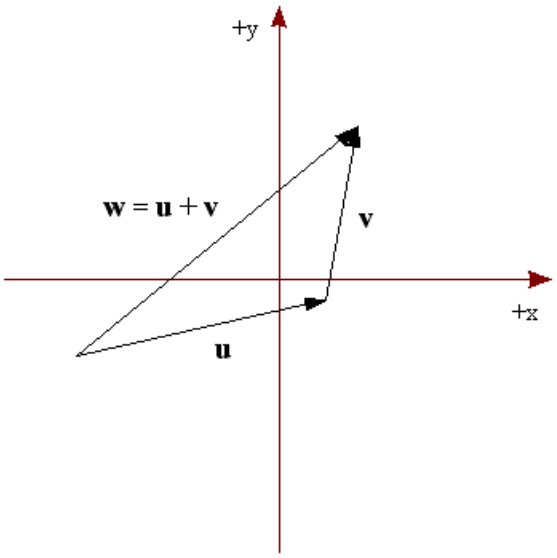


Figura 14 - w = u + v

Resta Vector-Vector

Se restan los elementos correspondientes de los dos vectores. Como en la suma, la resta de dos vectores sólo es posible si ambos son del mismo orden. Esto es,

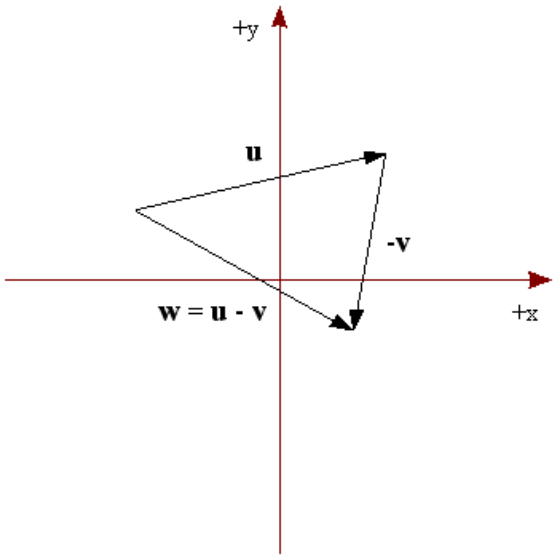


Figura 15 - w = u - v

u − **v** =

$(u_1, u_2, u_3, u_4, \ldots, u_n) - (v_1, v_2, v_3, v_4, \ldots, v_n) =$
$(u_1-v_1, u_2-v_2, u_3-v_3, u_4-v_4, \ldots, u_n-v_n)$

La resta entre vectores realmente es la suma del minuendo y el sustraendo en forma negativa. Esto es,

u − **v** = **u** + (−**v**)

Analizando esta operación a través de la geometría, vemos que implementamos la misma técnica de encadenar los vectores. Sin embargo, al tratarse de una suma pero con el vector sustraendo en el sentido contrario, encadenamos este vector negativo. Esto se puede ver mejor en la *Figura 15*.

Producto Escalar

Esta operación nos da como resultado un escalar que tiene relación con la magnitud de un vector resultante de dos vectores. El producto escalar se realiza multiplicando los componentes homólogos de ambos vectores y luego sumando tales productos. Por lo tanto, si tenemos que,

$$\mathbf{u} = (u_1, u_2, u_3, u_4, \ldots, u_n)$$
$$\mathbf{v} = (v_1, v_2, v_3, v_4, \ldots, v_n)$$

entonces su producto escalar es,

$$\mathbf{u}\cdot\mathbf{v} = u_1v_1 + u_2v_2 + u_3v_3 + u_4v_4 + \ldots + u_nv_n$$

El producto escalar tiene las siguientes propiedades:

- $\mathbf{u}\cdot\mathbf{u} = \|\mathbf{u}\|^2$
- $\mathbf{u}\cdot\mathbf{v} = \mathbf{v}\cdot\mathbf{u}$
- $\mathbf{u}\cdot(\mathbf{v}+\mathbf{w}) = \mathbf{u}\cdot\mathbf{v} + \mathbf{u}\cdot\mathbf{w}$
- $(s\mathbf{u})\cdot\mathbf{v} = s(\mathbf{u}\cdot\mathbf{v}) = \mathbf{u}\cdot(s\mathbf{v})$

Si *α* es el ángulo entre dos vectores, **u** y **v**, entonces tenemos que,

$$\mathbf{u}\cdot\mathbf{v} = \|\mathbf{u}\| \, \|\mathbf{v}\| \cos \alpha$$

Esto implica que existe una relación entre el ángulo α y el producto escalar. Con ambas fórmulas, podemos descubrir el ángulo α entre dos vectores. Esto se ve claramente en la *Figura 16*.

Podemos deducir los siguientes casos:

- **u**·**v** = 0 implica que el ángulo α = α/2 (90°) y por tanto, **u** y **v** son *perpendiculares*. Se dice que **u** y **v** son **ortogonales**. Si, además de ser ortogonales, **u** y **v** son vectores unitarios (de módulo 1), entonces decimos que los vectores son **ortonormales**.

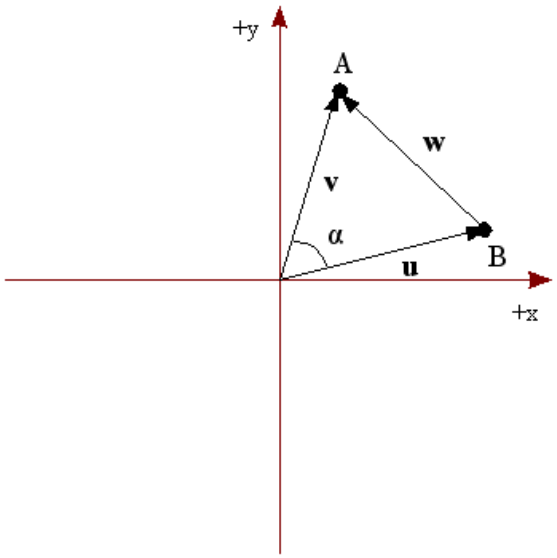


Figura 16 - Producto Escalar

- Si el ángulo α = 0 (0°) ó α = π (180°), entonces **u** y **v** son *paralelos*.

Aplicando la ley de cosenos, para triángulos,

$$c^2 = a^2 + b^2 - 2\,a\,b\cos\alpha$$

observamos la siguiente fórmula:

$$\|\mathbf{w}\|^2 = \|\mathbf{u}\|^2 + \|\mathbf{v}\|^2 - 2\,\|\mathbf{u}\|\,\|\mathbf{v}\|\cos\alpha$$

Esto implica que,

$$\|\mathbf{w}\|^2 = \|\mathbf{u}\|^2 + \|\mathbf{v}\|^2 - 2\,\mathbf{u}\cdot\mathbf{v}$$

Producto Vectorial

Esta operación sirve para determinar un nuevo vector que es perpendicular - ortogonal - a ambos vectores operandos. Dados dos vectores no paralelos, **u** y **v**, en un espacio tridimensional, el producto vectorial dará un tercer vector, **w**, que es ortogonal a ambos. Debemos tener:

$$\mathbf{w}\cdot\mathbf{u} = \mathbf{w}\cdot\mathbf{v} = 0$$

Tenemos dos vectores de tres elementos, **u** y **v**, definidos de la siguiente manera:

$$\mathbf{u} = (u_1, u_2, u_3)$$
$$\mathbf{v} = (v_1, v_2, v_3)$$

Definimos el vector **w** como el determinante con estos elementos:

$$\mathbf{w} = \mathbf{u}\times\mathbf{v} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \end{vmatrix} = (u_2v_3 - u_3v_2)\mathbf{i} - (u_1v_3 - u_3v_1)\mathbf{j} + (u_1v_2 - u_2v_1)\mathbf{k}$$

Aquí **i**, **j**, y **k** representan los vectores unitarios: (1, 0, 0), (0, 1, 0), y (0, 0, 1), respectivamente. Observamos que **i** es el vector con la misma dirección que el eje X, con el sentido positivo, y tiene como longitud 1. Lo mismo ocurre con los vectores **j** y **k**.

Podemos reescribir el resultado del determinante, que es el vector **w**:

$$\mathbf{w} = (u_2v_3 - u_3v_2,\ u_1v_3 - u_3v_1,\ u_1v_2 - u_2v_1)$$

El producto vectorial tiene las siguientes propiedades:

- $\mathbf{u}\times\mathbf{u} = \mathbf{u}\times\mathbf{0} = \mathbf{0}\times\mathbf{u} = \mathbf{0}$
- $\mathbf{u}\times\mathbf{v} = -(\mathbf{v}\times\mathbf{u})$

- $\mathbf{u} \times (\mathbf{v} + \mathbf{w}) = (\mathbf{u} \times \mathbf{v}) + (\mathbf{u} \times \mathbf{w})$
- $\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w}) = (\mathbf{u} \times \mathbf{v}) \cdot \mathbf{w}$
- $\mathbf{u} \times (\mathbf{v} \times \mathbf{w}) = (\mathbf{u} \cdot \mathbf{w})\,\mathbf{v} - (\mathbf{u} \cdot \mathbf{v})\,\mathbf{w}$
- $(s\mathbf{u}) \times \mathbf{v} = \mathbf{u} \times (s\mathbf{v}) = s\,(\mathbf{u} \times \mathbf{v})$,

  donde *s* es un número real: un escalar.
- $\|\mathbf{u} \times \mathbf{v}\|^2 = \|\mathbf{u}\|^2\,\|\mathbf{v}\|^2 - (\mathbf{u} \cdot \mathbf{v})^2$

Como se puede apreciar en la primera propiedad, el orden de los operandos para el producto vectorial influye en el vector resultante. Sin embargo, la única diferencia entre los dos posibles vectores, es un cambio de sentido; es decir:

$$\mathbf{u} \times \mathbf{v} = -(\mathbf{v} \times \mathbf{u})$$

Existe otra propiedad con el módulo del producto vectorial:

$$\|\mathbf{u} \times \mathbf{v}\| = \|\mathbf{u}\| * \|\mathbf{v}\| * \operatorname{sen} \alpha,$$

donde *α* es el ángulo formado entre los vectores **u** y **v**.
Dicho lo anterior, podemos deducir los siguientes casos:

- $\mathbf{u} \times \mathbf{v} = \mathbf{0}$

  implica que el ángulo α = 0 (0°) ó α = π (180°) y por tanto, **u** y **v** son *paralelos*.
- Si el ángulo α = π/2 (90°), entonces **u** y **v** son *perpendiculares*.

Ejemplos

Dados los vectores: **u** = (-3, 5, 1) y **v** = (4, -2, -1), determine el producto vectorial tanto de **u × v** como de **v × u**, y el ángulo entre ambos vectores, usando el módulo del producto vectorial.

$$\mathbf{w} = \mathbf{u} \times \mathbf{v} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ -3 & 5 & 1 \\ 4 & -2 & -1 \end{vmatrix}$$
$$= (5*(-1) - 1*(-2))\mathbf{i} - ((-3)*(-1) - 1*4)\mathbf{j} + ((-3)*(-2) - 5*4)\mathbf{k}$$
$$= -3\mathbf{i} + \mathbf{j} - 14\mathbf{k}$$

$$\mathbf{w'} = \mathbf{v} \times \mathbf{u} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 4 & -2 & -1 \\ -3 & 5 & 1 \end{vmatrix}$$
$$= ((-2)*1 - (-1)*5)\mathbf{i} - (4*1 - (-1)*(-3))\mathbf{j} + (4*5 - (-2)*(-3))\mathbf{k}$$
$$= 3\mathbf{i} - \mathbf{j} + 14\mathbf{k}$$

Como podemos observar **w** = **-w′**, por lo que ‖**w**‖ = ‖**w′**‖ :

$$\|\mathbf{w}\| = \sqrt{(-3)^2 + (1)^2 + (-14)^2} = \sqrt{206} = 14{,}3527 \text{ unidades}$$

Ahora podemos averiguar el ángulo α que forman estos dos vectores:

$$\|\mathbf{u}\| = \sqrt{(-3)^2 + (5)^2 + (1)^2} = \sqrt{35} = 5{,}9161 \text{ unidades}$$
$$\|\mathbf{v}\| = \sqrt{(4)^2 + (-1)^2 + (-1)^2} = \sqrt{18} = 4{,}2426 \text{ unidades}$$

$$\text{sen } \alpha = \frac{\|\mathbf{u} \times \mathbf{v}\|}{\|\mathbf{u}\| \, \|\mathbf{v}\|} = \frac{14{,}3527}{5{,}9161 * 4{,}2426} = 0{,}5718$$

$$\alpha = \text{sen}^{-1} \, 0{,}5718 = 0{,}6087 \text{ radianes} \Rightarrow 34{,}88°$$

Otras Operaciones

Aquí presentamos otras cuatro operaciones asociadas a vectores.

Combinación Lineal

Se trata de obtener un vector a partir de una combinación de otros *n* vectores $\mathbf{u_1}$, $\mathbf{u_2}$, $\mathbf{u_3}$, $\mathbf{u_4}$, …, $\mathbf{u_n}$ y un conjunto de *n* escalares a_1, a_2, a_3, a_4, …, a_n. Esto es,

$$\mathbf{u} = a_1\mathbf{u_1} + a_2\mathbf{u_2} + a_3\mathbf{u_3} + a_4\mathbf{u_4} + \ldots + a_n\mathbf{u_n}$$

Si el único conjunto de escalares tal que,

$$a_1\mathbf{u_1} + a_2\mathbf{u_2} + a_3\mathbf{u_3} + a_4\mathbf{u_4} + \ldots + a_n\mathbf{u_n} = \mathbf{0}$$

es

$$a_1 = a_2 = a_3 = a_4 = \ldots = a_n$$

entonces se dice que estos vectores son **linealmente independientes**. Dicho de otra manera, si **u** no puede ser representado usando una combinación lineal de otros vectores, entonces son linealmente independientes.

La mayor cantidad de vectores linealmente independientes que podemos encontrar para un espacio indica la **dimensión** de ese espacio. Por ejemplo, si tenemos tres vectores linealmente independientes entre sí, entonces ese espacio es tridimensional, ya que todos los vectores en ese espacio pueden ser representados por estos tres vectores linealmente independientes.

Base de vectores

Si un espacio vectorial tiene *n* dimensiones, entonces cualquier conjunto de *n* vectores linealmente independientes forma una base. Típicamente usamos una base de vectores para describir un *sistema de coordenadas* si estos vectores son perpendiculares entre sí. Esto implica que tales vectores y la base son *ortogonales*.

Digamos que tenemos el conjunto de vectores $\mathbf{v_1}$, $\mathbf{v_2}$, $\mathbf{v_3}$, $\mathbf{v_4}$, …, $\mathbf{v_n}$ que forma una base de vectores. Podemos representar cualquier vector **v** únicamente como una combinación lineal de esta base de vectores. Esto es,

$$\mathbf{v} = b_1\mathbf{v_1} + b_2\mathbf{v_2} + b_3\mathbf{v_3} + b_4\mathbf{v_4} + \ldots + b_n\mathbf{v_n}$$

donde b_1, b_2, b_3, b_4, …, b_n son escalares.

En la práctica, escribimos los vectores unitarios que forman la base del plano cartesiano así:

$\mathbf{i}$ y $\mathbf{j}$ para $\mathbb{R}^2$, e
$\mathbf{i}$ y $\mathbf{j}$ para $\mathbb{R}^3$

Nota: Los ingenieros escriben tales vectores con un circunflejo: $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$, y $\hat{\mathbf{k}}$.

Proyección

La proyección nos ayuda para determinar la longitud (escalar) de la proyección perpendicular de un vector **a** sobre la línea recta determinada por un vector **b**. También podemos hablar del componente de **a** sobre **b** como la proyección. Esto se puede calcular geométricamente de esta manera:

$$\text{comp}_{\mathbf{b}}\,\mathbf{a} = \|\mathbf{a}\| \cos\alpha$$

La expresión a la derecha del símbolo de igualdad se parece mucho a la definición del producto escalar. Por lo tanto, podemos reescribir la fórmula anterior de esta manera:

$$\text{comp}_{\mathbf{b}}\,\mathbf{a} = \frac{\|\mathbf{a}\|\,\|\mathbf{b}\|\cos\alpha}{\|\mathbf{b}\|} = \frac{\mathbf{a}\cdot\mathbf{b}}{\|\mathbf{b}\|}$$

El resultado es positivo si el ángulo $0 < \alpha < \pi/2$ y negativo si $\alpha > \pi/2$.

Ya que la proyección nos da la longitud de la proyección, podemos averiguar los vectores componentes de **a** sobre **b**. Esto es, podemos averiguar los vectores paralelo, $\mathbf{a}_{\parallel}$, y perpendicular, $\mathbf{a}_{\perp}$, de **a** con respecto al vector **b**. Algebraicamente, tenemos que,

$$\mathbf{a}_{\parallel} = (\text{comp}_{\mathbf{b}}\,\mathbf{a})\frac{\mathbf{b}}{\|\mathbf{b}\|}$$

$$\mathbf{a}_{\perp} = \mathbf{a} - \mathbf{a}_{\parallel}$$

Ejemplos

Algunos ejemplos acerca de estas operaciones son:

Ejemplo 1

Dados los siguientes vectores:

$$\mathbf{u} = (-3, 2)$$
$$\mathbf{v} = (-1, 0)$$
$$\mathbf{w} = (3, -2)$$

a. Determine cuáles de estos vectores son linealmente independientes entre ellos.
b. Averigüe cuáles de estos vectores forman una base.
c. Represente el vector que no es linealmente independiente y que no forma una base en términos de la base averiguada del apartado *b)*.

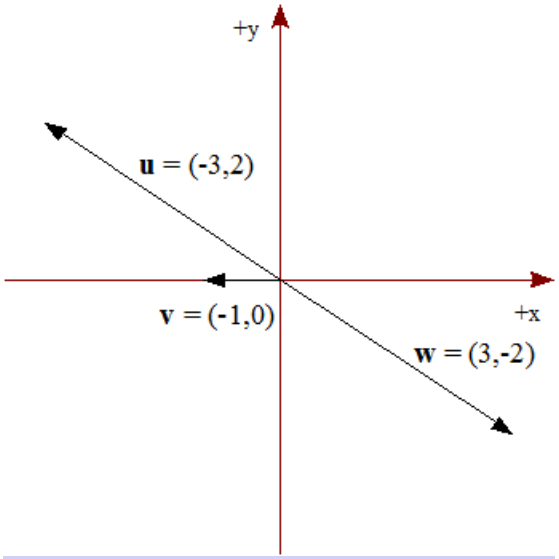


Figura 17 - Ejemplo 1

Solución

a. Comprobamos la combinación lineal de cada pareja de vectores por independencia lineal.

Veamos la pareja **u** y **v**,

$$\mathbf{0} = a\,\mathbf{u} + b\,\mathbf{v};$$
$$(0, 0) = a\,(-3, 2) + b\,(-1, 0) = (-3a-b, 2a)$$

Por lo que tenemos,

$$0 = -3a-b$$
$$0 = 2a \quad \Rightarrow \quad a = 0 \quad \Rightarrow \quad b = 0$$

Concluimos que **u** y **v** son linealmente independientes.
Ahora comprobamos la pareja **u** y **w**,

$$\mathbf{0} = a\,\mathbf{u} + b\,\mathbf{w};$$
$$(0, 0) = a\,(-3, 2) + b\,(3, -2) = (-3a+3b, 2a-2b)$$

Por lo que tenemos,

$$0 = -3a+3b$$
$$0 = 2a-2b$$

Esto implica que, a = b y por tanto, no existe una única solución.
Estos dos vectores son linealmente dependientes.
Por último, revisemos la pareja **v** y **w**,

$$\mathbf{0} = a\,\mathbf{v} + b\,\mathbf{w};$$
$$(0, 0) = a(-1, 0) + b(3, -2) = (-a+3b, -2b)$$

Por lo que tenemos,

$$0 = -a+3b \Rightarrow a = 0$$
$$0 = -2b \Rightarrow b = 0$$

Concluimos que **v** y **w** son linealmente independientes.

b. Podemos formar dos bases:
   1. **u** y **v**, y
   2. **v** y **w**

c. Escribimos cada vector que no pertence a la base como una combinación lineal de los otros dos vectores,
   1. Con la base, **u** y **v**, representamos **w** de la siguiente manera,

$$\mathbf{w} = a\,\mathbf{u} + b\,\mathbf{v};$$
$$(3, -2) = a(-3, 2) + b(-1, 0) = (-3a-b, 2a)$$

Por lo que tenemos,

$$3 = -3a-b$$
$$-2 = 2a \Rightarrow a = -1 \text{ y por tanto } b = 0$$

Es decir,

$$\mathbf{w} = -\mathbf{u}$$

   2. Usando la base, **v** y **w**, representamos **u** así,

$$\mathbf{u} = a\,\mathbf{v} + b\,\mathbf{w};$$
$$(-3, 2) = a(-1, 0) + b(3, -2) = (-a+3b, -2b)$$

Por lo que tenemos,

$$3 = -a+3b$$
$$2 = -2b \Rightarrow b = -1 \text{ y por tanto } a = 0$$

Al final, obtenemos que,

$$\mathbf{u} = -\mathbf{w}$$

Ejemplo 2

Dado el vector **u** = (2,4), determine sus componentes vectoriales sobre el vector, **v** = (3,-1).

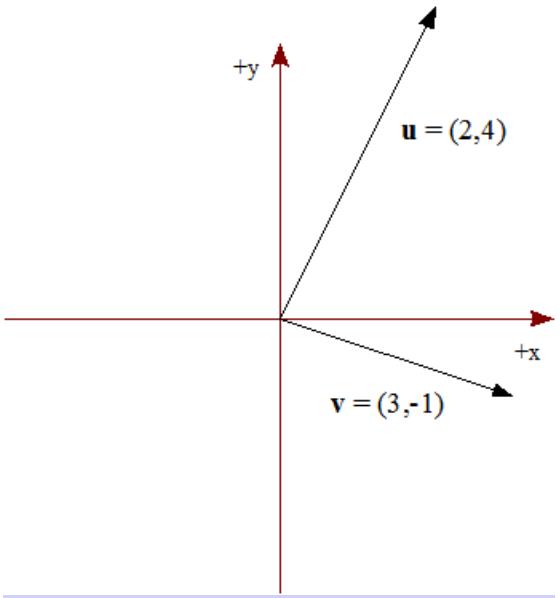


Figura 18 - Ejemplo 2

Solución

Calculemos el componente paralelo de **u**,

$$\text{comp}_{\mathbf{v}}\,\mathbf{u} = \frac{(2,4)\cdot(3,-1)}{\|\mathbf{v}\|} = \frac{6-4}{\sqrt{10}} = 0,6325$$

Obtenemos los siguientes vectores,

$$\mathbf{u}_{\parallel} = 0,6325*(3,-1) \,/\, \sqrt{10} = (3/5,-1/5) = (0,6,\ -0,2)$$

$$\mathbf{u}_{\perp} = \mathbf{u} - \mathbf{u}_{\parallel} = (2,4) - (0,6,\ -0,2) = (1,4,\ 4,2)$$

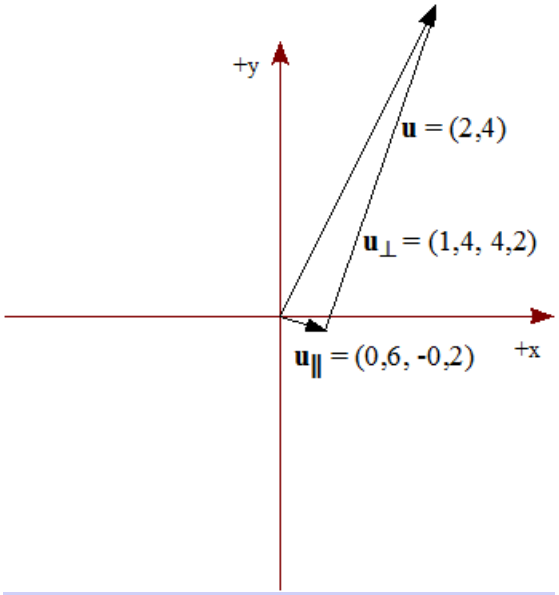


Figura 19 - Ejemplo 2 Solución

Apéndice 3: Matrices

Las matrices sirven para agrupar información en la forma de valores numéricos. Esta forma de agrupar valores tiene ciertas propiedades matemáticas. Para la teoría de gráficos, el mayor uso de matrices es en la representación de sistemas de coordenadas y numerosas transformaciones de tales. Generalmente, se usarán matrices que son 4 x 4, para las matrices de transformación, por ejemplo.

Definiciones

Una **matriz** es una lista de *n* x *m* números escalares, que se suele representar como *n* filas y *m* columnas. éstas son las **dimensiones** de la matriz: filas y columnas. Si *n* = *m*, entonces decimos que es una **matriz cuadrada** de dimensión (u orden) *n*. Los elementos de la matriz **A** son el conjunto de números escalares, $\{a_{ij}\}$, donde $i = 1, 2, \ldots, n$ y $j = 1, 2, \ldots, m$. Visto de otra manera:

$$\mathbf{A} = \begin{matrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{matrix}$$

Aquí **A** es una matriz de 4 x 4, o sea, una matriz cuadrada de dimensión (u orden) 4, ya que tiene 4 filas y 4 columnas.

La **traspuesta** de una matriz **A** de *n* x *m* es una matriz de *m* x *n* obtenida por el intercambio de las filas por las columnas de **A**. Esta matriz traspuesta se anota $\mathbf{A}^T$:

$$\mathbf{A^T} = \begin{matrix} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \end{matrix}$$

$$a_{13}\ a_{23}\ a_{33}\ a_{43}$$
$$a_{14}\ a_{24}\ a_{34}\ a_{44}$$

Operaciones de Matrices

Existen tres operaciones de matrices básicas: multiplicación escalar-matriz, suma matriz-matriz, y multiplicación matriz-matriz.

Multiplicación Escalar-Matriz

Es la multiplicación del escalar *s* por cada elemento de la matriz **A**.

$$s\mathbf{A} = \begin{matrix} sa_{11} & sa_{12} & sa_{13} & sa_{14} \\ sa_{21} & sa_{22} & sa_{23} & sa_{24} \\ sa_{31} & sa_{32} & sa_{33} & sa_{34} \\ sa_{41} & sa_{42} & sa_{43} & sa_{44} \end{matrix}$$

Este tipo de multiplicación tiene las propiedades conmutativa: s**A** = **A**s, y asociativa: s(t**A**) = (st)**A**.

Suma Matriz-Matriz

Se suman los elementos correspondientes de las dos matrices. La suma sólo se puede hacer si las dos matrices son del mismo orden: el número de filas y columnas es el mismo para la matriz **A** y **B**:

$$\mathbf{A} + \mathbf{B} = \begin{matrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} & a_{14}+b_{14} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} & a_{24}+b_{24} \\ a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} & a_{34}+b_{34} \\ a_{41}+b_{41} & a_{42}+b_{42} & a_{43}+b_{43} & a_{44}+b_{44} \end{matrix}$$

La suma de matrices acepta las propiedades conmutativa: **A** + **B** = **B** + **A**, y asociativa: **A** + (**B** + **C**) = (**A** + **B**) + **C**. Por supuesto, todas las matrices han de ser de orden *n* x *m*.

Multiplicación Matriz-Matriz

Se multiplican cada elemento de una fila de la primera matriz, **A,** de orden *n* x *r*, con su correspondiente elemento de la columna de la segunda matriz, **B**, de orden *r* x *m*. Cada producto es el sumando para la suma, que será el elemento resultante de la fila y columna intervenidas. Esto es mejor ilustrarlo:

$$\mathbf{A} = \begin{matrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{matrix} \quad \mathbf{B} = \begin{matrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{11} & b_{12} \end{matrix}$$

$$\mathbf{A} * \mathbf{B} = \begin{matrix} a_{11}*b_{11}+a_{12}*b_{21}+a_{13}*b_{31} & a_{11}*b_{12}+a_{12}*b_{22}+a_{13}*b_{32} \\ a_{21}*b_{11}+a_{22}*b_{21}+a_{23}*b_{31} & a_{21}*b_{12}+a_{22}*b_{22}+a_{23}*b_{32} \\ a_{31}*b_{11}+a_{32}*b_{21}+a_{33}*b_{31} & a_{31}*b_{12}+a_{32}*b_{22}+a_{33}*b_{32} \end{matrix}$$

Dicho de otro modo:

$$\mathbf{C} = \mathbf{A}*\mathbf{B} = [c_{ij}]$$

donde,

$$c_{ij} = \sum_{k=1}^{r} (a_{ik}*b_{kj})$$

La matriz resultante será **C**, que será de orden *n* x *m*. Por esta razón, sólo se pueden multiplicar matrices donde el número de columnas de la primera matriz sea el mismo que el de las filas de la segunda matriz. Es decir, la primera matriz de orden *s* x *t* se multiplica por la segunda matriz de orden *v* x *w*, entonces *t* = *v*, y la matriz resultante será de orden *s* x *w*.

La multiplicación de matrices acepta la propiedad asociativa: **A** * (**B** * **C**) = (**A** * **B**) * **C**, pero no siempre se da la conmutativa: **A** * **B** ≠ **B** * **A**. Por supuesto, todas las matrices han de ser de orden *n* x *m*.

Es importante mencionar la matriz identidad, que es una matriz cuadrada donde todos los elementos son ceros, a excepción de los elementos que se encuentran en la diagonal, donde son unos:

$$\mathbf{I} = \begin{matrix} 1 & 0 & 0 & 0 & \dots \\ 0 & 1 & 0 & 0 & \dots \\ 0 & 0 & 1 & 0 & \dots \\ 0 & 0 & 0 & 1 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{matrix} \qquad \mathbf{I} = [a_{ij}], \qquad a_{ij} = \begin{cases} 1, & \text{si } i = j \\ 0, & \text{de lo contrario} \end{cases}$$

La matriz identidad es una de las pocas matrices donde la propiedad conmutativa se puede aplicar: **I** * **A** = **A** * **I** = **A**. Cualquier matriz multiplicada por la identidad es igual a esa misma matriz: **I** * **A** = **A**. La matriz identidad es el elemento neutro para las matrices.

Ejemplos

$$\mathbf{A} = \begin{matrix} 3 & 3 & -1 & 0 \\ 0 & -2 & 2 & 0 \\ 1 & 0 & -2 & 0 \\ 2 & -3 & 4 & 1 \\ -1 & -1 & 0 & -2 \end{matrix} \qquad \mathbf{B} = \begin{matrix} 3 & 0 & 1 & 0 \\ -2 & -1 & -3 & 0 \\ -1 & -4 & 2 & 3 \\ 0 & 2 & 1 & 0 \\ 3 & 2 & 4 & 1 \end{matrix} \qquad \mathbf{C} = \begin{matrix} 2 & 0 & -1 & 1 \\ -1 & 2 & 0 & 2 \\ 0 & 0 & 2 & 0 \\ 1 & 1 & 2 & 1 \end{matrix}$$

A * **B** ⇒ No se puede, ya que **A** es de orden 5 x 4, y **B**, 5 x 4. El número de columnas de **A** no es igual al de filas de **B**. Lo mismo ocurre con **B** * **A**, y con **C** * **A**.

$$\mathbf{A} * \mathbf{C} = \begin{matrix} 3 & 6 & -5 & 9 \\ 2 & -4 & 4 & -4 \\ 2 & 0 & -5 & 1 \\ 8 & -5 & 8 & -3 \\ -3 & -4 & -3 & -4 \end{matrix} \qquad \mathbf{B} * \mathbf{C} = \begin{matrix} 6 & 0 & -1 & 3 \\ -3 & -2 & -2 & -4 \\ 5 & -5 & 11 & -6 \\ -2 & 4 & 2 & 4 \\ 5 & 5 & 7 & 8 \end{matrix} \qquad \mathbf{A} + \mathbf{B} = \begin{matrix} 6 & 3 & 0 & 0 \\ -2 & -3 & -1 & 0 \\ 0 & -4 & 0 & 3 \\ 2 & -1 & 5 & 1 \\ 2 & 1 & 4 & -1 \end{matrix}$$

Determinante de una Matriz

El determinante de una matriz cuadrada **A** es un escalar, y se representa así: **|A|** o det **A**.

Hay varias formas de calcular el determinante. La forma más general es siguiendo la regla de Cramer:

- El determinante de una matriz cuadrada de orden 2 es el siguiente:

$$\mathbf{A} = \begin{matrix} a & b \\ c & d \end{matrix} \qquad |\mathbf{A}| = a*d - b*c$$

- Con una matriz cuadrada de mayor orden, elegimos una columna o fila cualquiera. Ahora iremos cancelando la fila, por ejemplo, elegida y la primera columna. Los elementos restantes formarán otra matriz, y calcularemos el determinante de ésta. Este determinante se multiplicará por el elemento donde la fila elegida y primera columna se intersectan. Luego haremos lo mismo con la segunda columna y demás restantes. No se multiplica sin más, sino que hay un cambio de signo, dependiendo de cuál elemento se está multiplicando. Sigue este esquema, en forma de matriz:

```
+ − + − + − …
− + − + − + …
+ − + − + − …
− + − + − + …
+ − + − + − …
⋮ ⋮ ⋮ ⋮ ⋮ ⋮ ⋱
```

Ejemplos

Para entender cómo se realiza esto, es mejor ver un ejemplo:

```
     1  3 -1 -5          | 1  3 -1 -5 |
A =  3  2  0  1    |A| = | 3  2  0  1 |
    -3 -2 -1  0          |-3 -2 -1  0 |
     4  0  1  1          | 4  0  1  1 |
```

Ahora elegimos una fila o columna; elegiremos la 4ª fila. Por lo que el determinante será:

```
           | 3 -1 -5 |       | 1 -1 -5 |       | 1  3 -5 |
| 1  3 -1 |
|A| = -4 * | 2  0  1 | + 0 * | 3  0  1 | - 1 * | 3  2  1 | + 1 *
| 3  2  0 |
           |-2 -1  0 |       |-3 -1  0 |       |-3 -2  0 |
|-3 -2 -1 |
```

El primer producto se realiza mediante eliminando la 1ª columna y la 4ª fila. La intersección de éstas resulta en el elemento: 4, con el signo negativo, consultando el esquema visto anteriormente. El segundo producto aparece eliminando la 2ª columna y 4ª fila; el elemento de intersección es: 0, por lo que no hace falta ni calcular. El

tercer producto, es el determinante de los elementos restantes cuando se eliminan los elementos de la 3ª columna y 4ª fila, y se multiplica con la intersección: 1. Y el cuarto producto se hace de igual manera que los demás: 4ª columna y 4ª fila.

El cálculo de los tres determinantes se hace de igual forma: eligiendo una fila o columna, e ir eliminando hasta conseguir un determinante que sí sepamos hacer: a*b-d*c. Para una matriz de 3 x 3, aquí presentamos el determinante ya calculado:

```
    a b c
B = d e f    |B| = a*(ei-fh) - b*(di-fg) + c*(dh-eg)
    g h i
```

Volviendo a nuestro ejemplo, obtenemos los determinantes de las matrices de 3 x 3, para resolver la matriz de 4 x 4:

```
|A| = -4 * (3+2+10) - 1 * (2-9+0) + 1 * (-2+9+0)
    = (-4)*15 + (-1)*(-7) + (1)*(7)
    = -60 + 7 + 7
    = -46
```

Propiedades

Existen las siguientes propiedades acerca de las matrices y sus determinantes:

1. Si dos filas (o dos columnas) de una matriz cuadrada, **A**, son idénticas, entonces det **A** = 0.
2. Si dos filas (o dos columnas) de una matriz cuadrada, **A**, son intercambiables, entonces sólo el signo de det **A** es cambiado.
3. El valor de det **A** no varía si se suma un múltiplo de una fila a otra fila o si se suma un múltiplo de una columna a otra columna.
4. Si tenemos dos matrices cuadradas, **A** y **B**, del mismo orden, entonces det (**AB**) = (det **A**) (det **B**).
5. Si una matriz, **A**, es invertible, entonces det ($\mathbf{A}^{-1}$) = (det **A**)$^{-1}$.
6. Si una matriz, **A**, es cuadrada de orden *n*, entonces det (k**A**) = k^n(det **A**), donde k es un escalar.

Inversión de una Matriz

Para invertir una matriz, ésta debe ser una matriz cuadrada. Si,
q = **Ap**,
queremos saber si podemos encontrar una matriz cuadrada **B** tal que
p = **Bq**.
Sustituyendo por **q**,
p = **Bq** = **BAp**.
Haciendo un cambio de notación,
p = **Ip**.
Para que,
Ip = **BAp**, entonces
I = **BA**.

Si tal matriz **B** existe, entonces es la inversa a **A**, y **A** la denominamos *no singular*. También podemos decir que una matriz no invertible es *singular*. La inversa a **A** se escribe $\mathbf{A}^{-1}$. El resultado fundamental sobre inversas es el siguiente: La inversa a una matriz cuadrada existe si y sólo si el determinante de la matriz es distinto a cero. Podemos calcular la inversa usando determinantes o usando razonamiento geométrico. Por ejemplo, la inversa de una matriz con propiedades de simetría geométrica, es exactamente el reverso de esa matriz; como es el caso de rotaciones por un eje, que contiene senos y cosenos de un mismo ángulo.

Método

Usaremos el método de Cramer. Si D es el determinante de **A**, entonces **B** - la inversa de **A** - está formada por b_{ij}, donde cada b_{ij} es el determinante de la submatriz, creada por la eliminación de la fila *i* y de la columna *j* de la matriz $\mathbf{A}^{\mathbf{T}}$, dividido entre D. Esto es mejor explicarlo con un ejemplo.

Ejemplos

$$\mathbf{A} = \begin{vmatrix} 1 & 3 & -1 \\ 3 & 2 & 0 \\ -3 & -2 & -1 \end{vmatrix} \quad |\mathbf{A}| = \begin{vmatrix} 1 & 3 & -1 \\ 3 & 2 & 0 \\ -3 & -2 & -1 \end{vmatrix} = 7 = D \neq 0 \Rightarrow \mathbf{A} \text{ es invertible (o no singular)}$$

$$\mathbf{A^T} = \begin{vmatrix} 1 & 3 & -3 \\ 3 & 2 & -2 \\ -1 & 0 & -1 \end{vmatrix}$$

$\mathbf{A^{-1}} = \mathbf{B} = [b_{ij}]$,

$$b_{11} = \frac{\begin{vmatrix} 2 & -2 \\ 0 & -1 \end{vmatrix}}{D} = -0,2857 \qquad b_{12} = \frac{-\begin{vmatrix} 3 & -2 \\ -1 & -1 \end{vmatrix}}{D} = 0,7143$$

$$b_{13} = \frac{\begin{vmatrix} 3 & 2 \\ -1 & 0 \end{vmatrix}}{D} = 0,2857$$

$$b_{21} = \frac{-\begin{vmatrix} 3 & -3 \\ 0 & -1 \end{vmatrix}}{7} = 0,4286 \qquad b_{22} = \frac{\begin{vmatrix} 1 & -3 \\ -1 & -1 \end{vmatrix}}{7} = -0,5714$$

$$b_{23} = \frac{-\begin{vmatrix} 1 & 3 \\ -1 & 0 \end{vmatrix}}{7} = -0,4286$$

$$b_{31} = \frac{\begin{vmatrix} 3 & -3 \\ 2 & -2 \end{vmatrix}}{7} = 0 \qquad b_{32} = \frac{-\begin{vmatrix} 1 & -3 \\ 3 & -2 \end{vmatrix}}{7} = -1$$

$$b_{33} = \frac{\begin{vmatrix} 1 & 3 \\ 3 & 2 \end{vmatrix}}{7} = -1$$

Como podemos ver, b_{11} se calcula dividiendo dos determinantes. El numerador se calcula a partir de la matriz que queda eliminando la primera fila y columna de la matriz $\mathbf{A^T}$. El elemento b_{12} se calcula de la misma manera. Creamos la matriz del numerador eliminando la

primera fila y la segunda columna de la matriz $\mathbf{A}^{\mathbf{T}}$. Y así sucesivamente. También aplicamos los signos positivo o negativo siguiendo la misma lógica que empleamos cuando calculamos el determinante de una matriz.

Al final, obtenemos $\mathbf{A}^{-1}$ que es:

$$\mathbf{B} = \mathbf{A}^{-1} = \begin{matrix} -0,2857 & -0,7143 & 0,2857 \\ -0,4286 & -0,5714 & 0,4286 \\ 0 & -1 & -1 \end{matrix}$$

Se puede comprobar, fácilmente, que **AB** = **I**

Cambio de Representación

Usando matrices con los vectores base podemos cambiar la representación de cualquier conjunto de vectores (no base). Supongamos que tenemos dos vectores de dimensión *n* que forman la base del espacio vectorial: $\{u_1, u_2, u_3, \dots, u_n\}$ y $\{v_1, v_2, v_3, \dots, v_n\}$. Por ejemplo, el vector **w** puede ser expresado como:

$\mathbf{w} = a_1u_1 + a_2u_2 + a_3u_3 + \dots + a_nu_n$,

o

$\mathbf{w'} = b_1v_1 + b_2v_2 + b_3v_3 + \dots + b_nv_n$

Veamos cómo convertimos de la representación de **w** a la de **w′**. Los vectores base $\{v_1, v_2, v_3, \dots, v_n\}$ pueden expresarse como vectores en la base $\{u_1, u_2, u_3, \dots, u_n\}$. Por esto, existe un conjunto de escalares α_{ij} tal que:

$$\begin{matrix} u_1 \\ u_2 \\ u_3 \\ . \\ . \\ . \\ u_n \end{matrix} = \mathbf{A} \begin{matrix} v_1 \\ v_2 \\ v_3 \\ . \\ . \\ . \\ v_n \end{matrix}$$

Donde **A** es una matriz *n* x *n* : $\mathbf{A} = [\alpha_{ij}]$

Podemos usar matrices en columna para representar ambos vectores, **w** y **w′**, como:

$$\mathbf{w} = \mathbf{a}^T \begin{matrix} u_1 \\ u_2 \\ u_3 \\ . \\ . \\ . \\ u_n \end{matrix} \quad \text{Donde } \mathbf{a} = [a_i],$$

Definamos **b** como

$$\mathbf{b} = [b_i],$$

y podemos escribir **w′** como

$$\mathbf{w'} = \mathbf{b}^T \begin{matrix} v_1 \\ v_2 \\ v_3 \\ . \\ . \\ . \\ v_n \end{matrix}$$

Sustituyendo, obtenemos que

$$\mathbf{b}^T = \mathbf{a}^T \mathbf{A}$$

Ejemplos

Supongamos que tenemos un vector **w** = (3, -2, 1), representado en la base $\mathbf{u_1}$ = (1, 0, 0), $\mathbf{u_2}$ = (0, 1, 0), y $\mathbf{u_3}$ = (0, 0, 1). Queremos hallar el vector **w′**, que es igual a **w**, pero representado en otra base: $\mathbf{v_1}$ = (1, 0, 0), $\mathbf{v_2}$ = (1, 1, 0), and $\mathbf{v_3}$ = (1, 1, 1).

Representamos **w** en términos de la base de vectores **u**:

$$\mathbf{w} = 3\mathbf{u_1} - 2\mathbf{u_2} + 1\mathbf{u_3}$$

por lo que,

$\mathbf{a} = [3\ -2\ 1]$.

Representando **w′** en términos de **v**, obtenemos:

$\mathbf{w'} = b_1v_1 + b_2v_2 + b_3v_3$

Ahora calculamos **A**, que representa la matriz de la representación de la base de vectores **v**, en términos de **u**:

$v_1 = \mathbf{u_1}$
$v_2 = \mathbf{u_1} + \mathbf{u_2}$
$v_2 = \mathbf{u_1} + \mathbf{u_2} + \mathbf{u_3}$

por lo que,

$$\mathbf{A} = \mathbf{B^{-1}} = \begin{matrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{matrix}^{-1} = \begin{matrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{matrix}$$

Ahora averiguamos **b**:

$$\mathbf{b} = [3\ -2\ \ 1] * \begin{matrix} 1 & -1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{matrix} = [3\ -5\ 3]$$

Sabiendo **b**, podemos calcular **w′**:

$\mathbf{w'} = 3\mathbf{v_1} - 5\mathbf{v_2} + 3\mathbf{v_3}$

Producto Escalar

Dados dos vectores no perpendiculares, **u** y **v**, el producto escalar dará un escalar (número real). Si los vectores, **u** y **v**, contienen los siguientes elementos: u_1, u_2, u_3, …, u_n, y v_1, v_2,v_3, …, v_n, respectivamente, entonces definimos el producto escalar de esta forma:

$\mathbf{u} \bullet \mathbf{v} = (u_1, u_2, u_3, \ldots, u_n) \bullet (v_1, v_2, v_3, \ldots, v_n)$

$= u_1*v_1 + u_2*v_2 + u_3*v_3 + \ldots + u_n*v_n$
$= s$, donde s es un escalar.

Existe una relación con el módulo del producto escalar y los de los vectores con el coseno del ángulo formado con dichos vectores:

$\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| * \|\mathbf{v}\| * \cos \alpha$,

donde α es el ángulo formado entre los vectores **u** y **v**.

Ejemplos

Si **u** = (-1, 2, 5, 3, 0) y **v** = (-3, -2, 4, 0, -1), entonces

$$\begin{aligned}\mathbf{u} \cdot \mathbf{v} &= (-1, 2, 5, 3, 0) \cdot (-3, -2, 4, 0, -1) \\ &= (-1)*(-3) + 2*(-2) + 5*4 + 3*0 + 0*(-1) \\ &= 3 - 4 + 20 \\ &= 19\end{aligned}$$

Ahora podemos averiguar el ángulo entre estos vectores:

$$\begin{aligned}\|\mathbf{u}\| &= \sqrt{(-1)^2 + (2)^2 + (5)^2 + (3)^2 + (0)^2} \\ &= \sqrt{39} \cong 6{,}2450 \text{ unidades} \\ \|\mathbf{v}\| &= \sqrt{(-3)^2 + (-2)^2 + (4)^2 + (0)^2 + (-1)^2} \\ &= \sqrt{30} \cong 5{,}4472 \text{ unidades}\end{aligned}$$

$$\cos \alpha = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| * \|\mathbf{v}\|} = \frac{19}{6{,}2450 * 5{,}4472} \cong 0{,}5555$$

$$\alpha = \cos^{-1} 0{,}5555 = 0{,}9819 \text{ radianes} \cong 56{,}26°$$

Producto Vectorial

Dados dos vectores no paralelos, **u** y **v**, en un espacio tridimensional, el producto vectorial dará un tercer vector, **w**, que es ortogonal a ambos. Debemos tener:

$\mathbf{w} \cdot \mathbf{u} = \mathbf{w} \cdot \mathbf{v} = 0$.

Si los vectores, **u** y **v**, contienen los siguientes elementos: u_1, u_2, u_3, y v_1, v_2, v_3, respectivamente, entonces definimos **w** como el determinante con estos elementos:

$$\mathbf{w} = \mathbf{u} \times \mathbf{v} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ u_1 & u_2 & u_3 \\ v_1 & v_2 & v_3 \end{vmatrix} = \mathbf{i}(u_2 v_3 - u_3 v_2) - \mathbf{j}(u_1 v_3 - u_3 v_1) + \mathbf{k}(u_1 v_2 - u_2 v_1)$$

Aquí **i**, **j**, y **k** representan los vectores unitarios: (1, 0, 0), (0, 1, 0), y (0, 0, 1), respectivamente. Observamos que **i** es el vector con la misma dirección que el eje X, con el sentido positivo, y tiene como longitud 1, de ahí el nombre "unitario". Lo mismo ocurre con los vectores **j** y **k**, y los ejes Y y Z, respectivamente.

Podemos reescribir el resultado del determinante, que es el vector **w**:

$$\mathbf{w} = \begin{matrix} u_2 v_3 - u_3 v_2 \\ u_3 v_1 - u_1 v_3 \\ u_1 v_2 - u_2 v_1 \end{matrix}$$

El orden de los operandos para el producto vectorial influye en el vector resultante. Sin embargo, la única diferencia entre los dos posibles vectores, es un cambio de sentido; es decir:

$$\mathbf{u} \times \mathbf{v} = -(\mathbf{v} \times \mathbf{u})$$

Existe otra propiedad con el módulo del producto vectorial:

$$\|\mathbf{u} \times \mathbf{v}\| = \|\mathbf{u}\| * \|\mathbf{v}\| * \operatorname{sen} \alpha,$$

donde α es el ángulo formado entre los vectores **u** y **v**.

Ejemplos

u = (-3, 5, 1), y **v** = (4, -2, -1)

$$\begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \end{vmatrix}$$

$$\mathbf{w} = \mathbf{u} \times \mathbf{v} = \begin{vmatrix} -3 & 5 & 1 \\ 4 & -2 & -1 \end{vmatrix}$$
$$= \mathbf{i}(5*(-1) - (1*(-2))) - \mathbf{j}((-3)*(-1) - 1*4) + \mathbf{k}((-3)*(-2) - 5*4)$$
$$= -3\mathbf{i} + \mathbf{j} + (-14)\mathbf{k}$$

$$\mathbf{w'} = \mathbf{v} \times \mathbf{u} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 4 & -2 & -1 \\ -3 & 5 & 1 \end{vmatrix}$$
$$= \mathbf{i}(1*(-2) - (5*(-1))) - \mathbf{j}(1*4 - (-3)*(-1)) + \mathbf{k}(5*4 - (-3)*(-2))$$
$$= 3\mathbf{i} - \mathbf{j} + 14\mathbf{k}$$

Como podemos observar **w** = **-w'**, por lo que ‖**w**‖ = ‖**w'**‖ :

$$\|\mathbf{w}\| = \sqrt{(-3)^2 + (1)^2 + (-14)^2}$$
$$= \sqrt{206} \cong 14,3527 \text{ unidades}$$

Ahora podemos averiguar el ángulo que forman estos dos vectores:

$$\|\mathbf{u}\| = \sqrt{(-3)^2 + (5)^2 + (1)^2} = \sqrt{35} \cong 5,9161 \text{ unidades}$$
$$\|\mathbf{v}\| = \sqrt{(4)^2 + (-1)^2 + (-1)^2} = \sqrt{18} \cong 4,2426 \text{ unidades}$$

$$\text{sen } \alpha = \frac{\|\mathbf{u} \times \mathbf{v}\|}{\|\mathbf{u}\| * \|\mathbf{v}\|} = \frac{14,3527}{5,9161 * 4,2426} \cong 0,5718$$

$$\alpha = \text{sen}^{-1}\ 0,5718 = 0,6087 \text{ radianes} \cong 34,88°$$

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 escritos en C: *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 bibliotecas 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.

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 los 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.

Capítulo 4

No existen nuevas funciones ni macros.

Capítulo 5

No existen nuevas funciones ni macros.

Capítulo 6

No existen nuevas funciones ni macros.

Capítulo 7

No existen nuevas funciones ni macros.

Bibliografía

1. James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes. Computer Graphics: Principles and Practice. Boston, MA: Addison-Wesley, 1996
2. Edward Angel. Interactive Computer Graphics: A Top-Down Approach with OpenGL™. Reading, MA: Addison Wesley Longman, 1997
3. James Ward Brown, Ruel V. Churchill. Complex Variables and Applications. New York: McGraw-Hill, 1996
4. Bjarne Stroustrup. The C++ Programming Language. Reading, MA: Addison Wesley Longman, 1997
5. Scott Robert Ladd. C++ Simulations and Cellular Automata. New York: M&T Books, 1995
6. Timothy Wegner, Mark Peterson. Fractal Creations: Explore the Magic of Fractals on your PC. Mill Valley, CA: The Waite Group, 1991
7. Clayton Walnum. 3-D Graphics Programming with OpenGL™. Indianapolis, IN: QUE, 1995
8. Kurt Gieck, Reiner Gieck. Engineering Formulas. New York: McGraw-Hill, 1990
9. Christopher Clapham. The Concise Oxford Dictionary of Mathematics. Oxford: The Oxford University Press, 1990
10. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. Introduction to Algorithms. Cambridge, MA: The MIT Press, 1990. New York: McGraw-Hill Book Company, [c. 1990]
11. Adrian Perez, with Dan Royer. Advanced 3-D Game Programming Using DirectX® 7.0. Plano, TX: Wordware, 2000
12. John de Goes. 3D Game Programming with C++. Scottsdale, AZ: The Coriolis Group, 2000
13. Roger T. Stevens. Fractal Programming and Ray Tracing with C++. San Mateo, CA: M&T, 1991
14. André LaMothe. Tricks of the Windows Game Programming Gurus: Fundamentals of 2D and 3D Game Programming. Indianapolis, IN: Sams, 1999

15. Alexander Enzmann, Lutz Kretzschmar, Chris Young. Ray Tracing Worlds with POV-Ray. Corte Madera, CA: Waite Group Press, 1994
16. Uwe Braun. Atari® ST™: 3-D Graphics Programming. n.p. Abacus Software, 1988
17. Ian O. Angell. High-Resolution Computer Graphics Using C. London: MacMillan, 1990. New York: Halsted Press, [c. 1990]
18. Christopher D. Watkins, Larry Sharp. Programming in 3 Dimensions: 3-D Graphics, Ray Tracing, and Animation. San Mateo, CA: M&T Books, 1992
19. Richard Haberman. Elementary Applied Partial Differential Equations with Fourier Series and Boundary Value Problems. Upper Saddle River, NJ: Prentice Hall, 1998
20. John de Goes. Cutting-Edge 3D Game Programming with C++. Scottsdale, AZ: Coriolis, 1996
21. Conrac Division, Conrac Corporation. Raster Graphics Handbook. New York: Van Nostrand Reinhold, 1985
22. Cay S. Horstmann, Gary Cornell. Core Java™ 2: Volume I - Fundamentals. Upper Saddle River, NJ: Prentice Hall, 2003
23. Cay S. Horstmann, Gary Cornell. Core Java™ 2: Volume II - Advanced Features. Upper Saddle River, NJ: Prentice Hall, 2002
24. Roger T. Stevens, Christopher D. Watkins. Advanced Graphics Programming in Turbo Pascal. Redwood City, CA: M&T, 1991
25. David H. Eberly. 3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics. San Francisco, CA: Morgan Kaufmann, 2006

Tabla de contenido