Tema XI

Visualización standard: obtención de la imagen discreta

Ricardo Ramos

Colaboradores: José Carlos Fernández Pérez, Juan Fernando Pinto Maudo, Carmen Fernández Campo, Juan Carlos Rodríguez Rodríguez, Eduardo Alonso Martínez, Ignacio Vera Mariño, Ángel Ojeda de la Vega

En este tema se verá todo lo referente a la discretización de las imágenes vectoriales. Comenzaremos con las opciones de discretización, cómo se rasterizan las aristas y polígonos, las técnicas incrementales de aplicación de la intensidad, la eliminación de líneas y caras ocultas mediante el Z-buffer, y finalmente el pseudocódigo general del algoritmo de rendering standard.

1. Introducción

La segunda (y última) fase del proceso de rendering standard consiste en transformar la imagen vectorial en una imagen discreta, que es la que aparece en los monitores. Al proceso que traslada la información visual del espacio bidimensional continuo (imagen vectorial) hacia el espacio bidimensional discreto (imagen raster) se conoce como **discretización**.

Cuando se trabaja con modelos poliédricos, *la discretización de la imagen vectorial se reduce a discretizar los polígonos que aparecen en ella.*

En la **discretización de los polígonos** se distinguen claramente varios subprocesos, que normalmente se analizan por separado, aunque en su ejecución están estrechamente relacionados. Por un lado, se han de localizar los puntos (píxeles) de las fronteras del polígono (**rasterización de aristas**), y los puntos del interior (**rasterización de superficies**)[1].

Además, es necesario averiguar si los píxeles rasterizados son o no visibles[2]. De esto se encargan los **algoritmos de eliminación de caras ocultas**.

---

[1] Muchos textos, cuando hablan de la "rasterización de polígonos", se refieren exclusivamente a la rasterización de la superficie.

[2] El proceso de culling no garantiza que los polígonos candidatos a "visibles" realmente lo sean.

Finalmente, se ha de dar color a los píxeles (de interior o frontera) que sean visibles. Para esto se utilizan las **técnicas o métodos de aplicación de intensidad**, conocidos también como *métodos incrementales de intensidad*.

Los procesos anteriores proporcionan imágenes discretas de calidad aceptable, pero no suficiente. Si se desea aumentar la calidad, se han de ejecutar procesos adicionales que proporcionen sombras, texturas, etc. Tradicionalmente estos procesos no son considerados como integrantes del proceso de discretización, aunque no existe una razón convincente para no hacerlo.

Cuando finaliza la generación de la imagen vectorial, se dispone de las coordenadas de los vértices de los polígonos, dadas en el SRPV. Esta será la información inicial para la discretización de los polígonos, aunque no la única, ya que también han de estar disponibles los datos suficientes para determinar la visibilidad de los polígonos, y dar color a los píxeles.

Los preparativos y procesos para la obtención de la imagen raster se ejecutan a lo largo de toda la cadena de visualización, es decir, comienzan antes de finalizar la generación de la imagen vectorial. Para facilitar su estudio, los procesos de discretización se mostrarán de manera independiente, desvinculados del proceso global de rendering. Al final del tema veremos cómo y cuándo encaja cada proceso dentro del proceso general.

2. Discretización de polígonos

Como acabamos de ver, discretizar un polígono vectorial consiste en trasladar éste al espacio discreto bidimensional, intentando mantener la mayor semejanza posible con el original. En la discretización de los polígonos se han de realizar tres tareas básicas:

1) localizar en el espacio discreto los píxeles correspondientes al contorno e interior de los polígonos (*rasterización del polígono*),

2) averiguar si los píxeles localizados son visibles o no, y

3) calcular la intensidad en los píxeles visibles

Normalmente, los puntos de frontera son bastante más difíciles de localizar que los de interior, requiriéndose la utilización de *algoritmos trazadores de líneas,* según sea la técnica de rasterización que se adopte. Por tal motivo, el proceso de rasterización de los contornos tiene entidad propia, denominándose comúnmente como proceso de *rasterización de aristas.* En cambio, *para encontrar los píxeles de interior, con un simple incremento suele ser suficiente*. Esto hace que el proceso de *rasterización de las superficies* pase inadvertido, integrado por lo común en otros procesos.

La discretización de los polígonos *puede hacerse siguiendo el sentido marcado por las líneas de rastreo* (líneas raster o líneas scan), aunque también es posible discretizar independientemente de ellas.

Orden de renderizado de polígonos

A la hora de renderizar los polígonos se suele adoptar uno de estos tres planteamientos:

1) *renderizar polígono a polígono*

2) *renderizar dos o más polígonos a la vez (renderizado múltiple), siguiendo la línea scan.*

3) *efectuar un renderizado híbrido de los anteriores.*

El renderizado polígono a polígono *consiste en discretizar cada polígono de la imagen vectorial de modo aislado*, o sea, sin tener en cuenta los polígonos restantes.

Con respecto al renderizado múltiple, *lo que se hace es ir discretizando todos los polígonos de la imagen vectorial que son cruzados por una determinada línea de rastreo.*

Como era de suponer, estos dos planteamientos básicos de renderizado tienen sus pros y contras. Así, una de las mayores ventajas del renderizado polígono a polígono es que *no impone limitaciones a la complejidad de la escena vectorial*, puesto que sólo mantiene en memoria la información de un polígono. Por contra, no aprovecha la información compartida entre polígonos, como las aristas comunes a los polígonos adyacentes, lo que resta eficiencia al proceso.

Por su lado, el renderizado múltiple, siguiendo el orden de las líneas scan, tiene la ventaja de permitir una mayor implantación hardware del proceso. Además, saca partido de la información compartida por los polígonos, lo que facilita, entre otras, las operaciones anti-aliasing. A cambio, ha de mantener en memoria toda la información de discretización, sombreado y textura de cada uno de los polígonos que cruza la línea de rastreo, lo que impone un tope a la complejidad de la imagen, ya que la memoria es limitada.

A menudo, cuando existen dos o más planteamientos alternativos, se puede pensar en una solución intermedia (híbrida). En este caso, los polígonos de la imagen vectorial pueden agruparse según los objetos independientes (separados) a que pertenezcan. Seguidamente, se efectúa de modo individual (*objeto a objeto*), el discretizado múltiple de los polígonos de cada grupo (objeto). Así, se puede aprovechar la información compartida por los polígonos de cada objeto, y además, el límite de complejidad estará impuesto por el objeto con mayor número de polígonos, no por el total de polígonos en el escenario, dejando así mayor margen de maniobra.

Según sea el orden de renderizado que se adopte, los métodos de discretización difieren. Por tanto, veremos cómo son las técnicas y procesos en cada uno de los planteamientos anteriores, aunque haremos hincapié en la discretización polígono a polígono, por ser más pedagógica.

2.1 Discretización polígono a polígono

Hay dos formas básicas de llevar a cabo la discretización polígono a polígono:

- *método de sembrado,* en el cual, partiendo de un punto que se sabe que está en el interior del polígono (*semilla*), se buscan sus puntos adyacentes. Cuando un punto adyacente no pertenece al interior del polígono, entonces pertenece a la frontera (figura 1). En caso contrario, se toma como punto de partida para volver a ejecutar el algoritmo y así, recursivamente, se llegará a encontrar todos los puntos del interior y de frontera. Este método es apropiado cuando los polígonos se discretizan de modo interactivo, es decir, cuando el usuario señala (p.e., mediante el ratón) un punto interno de un polígono (semilla), para que éste sea rellenado.

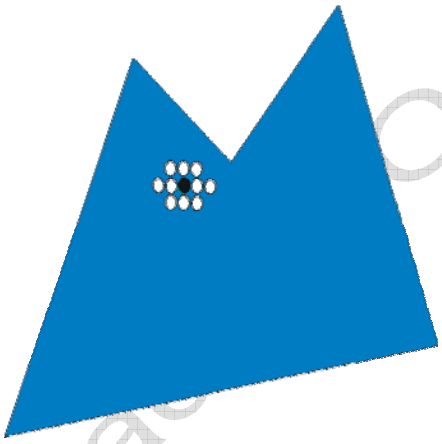


figura 1: método del sembrado para rellenar los polígonos

- *método de la línea scan,* que consiste en discretizar los polígonos según el orden de recorrido de las líneas scan. Este método puede considerarse como un caso extremo del renderizado híbrido (objeto a objeto), cuando los objetos se reducen a simples polígonos. Es, con diferencia, el método más utilizado, y por tanto, el que veremos con mayor detenimiento.

2.1.1 Discretización siguiendo las líneas scan

A) Información de discretización

Dado que los polígonos se discretizan siguiendo el orden marcado por las líneas scan, un concepto básico en este método es el de **segmento raster** (*o de rasterización*). En un polígono dado, dichos segmentos quedan definidos por los *puntos de interior, situados entre dos puntos de frontera, que están en una misma línea raster.*

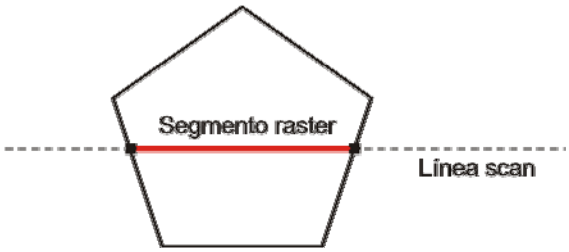


figura 2: segmento raster

En el fondo, lo que se hace al discretizar los polígonos siguiendo la línea scan, no es otra cosa que *organizar los polígonos como conjuntos de segmentos raster*. Para ello, *se ha de disponer de la información necesaria para discretizar cada segmento raster del polígono*. Dicha información ha de permitir:

1) localizar el segmento raster,

2) averiguar qué píxeles del segmento son visibles y cuáles no, y

3) calcular la intensidad y color de los píxeles visibles del segmento raster.

Vemos que los pasos para discretizar un segmento son los mismos que para discretizar un polígono, aunque *la información requerida es más simple*.

Para localizar los segmentos raster es necesario conocer las coordenadas inicial y final del segmento, en el SRPV. Dada una línea scan *"s"*, las coordenadas iniciales y finales suelen indicarse por (x_i, y_s) y (x_f, y_s), respectivamente.

La información necesaria para averiguar si los píxeles son visibles o no, depende del método utilizado. Entre los muchos algoritmos que hay para eliminar las caras y líneas ocultas, el más utilizado en la discretización polígono a polígono es el conocido como **Z-buffer**, por lo que será tratado con mayor detenimiento. Para este algoritmo es necesario conocer la **profundidad** en cada extremo del segmento. Ésta viene dada por *las coordenadas Z (en el SRV) de los puntos de las aristas, cuyas proyecciones determinan los extremos de los segmentos raster*.

En cuanto al cálculo de la intensidad y color de los píxeles en los segmentos raster, pronto veremos que son dos las opciones básicas que producen resultados satisfactorios: a) *disponer de la intensidad* de luz que llega a los extremos del segmento o, b) *conocer la normal* a la superficie en dichos extremos.

La información de discretización la proporcionan los **procesos de discretización**, que veremos más abajo.

B) Registro de la información de discretización

La mayoría de los libros de texto que tratan la discretización scan de los polígonos lo hacen exponiendo el *método general*. Lo de "general" significa que con dicho método es posible discretizar todo tipo de polígonos, o sea, cóncavos, convexos, e incluso aquellos que tengan agujeros en su interior.

La diferencia principal entre el método general de discretización y otros métodos más restrictivos *radica en el modo organizar la información de discretización de cada polígono.* En la figura 3 se muestra la estructura de datos que utiliza en el método general para registrar la información de discretización.

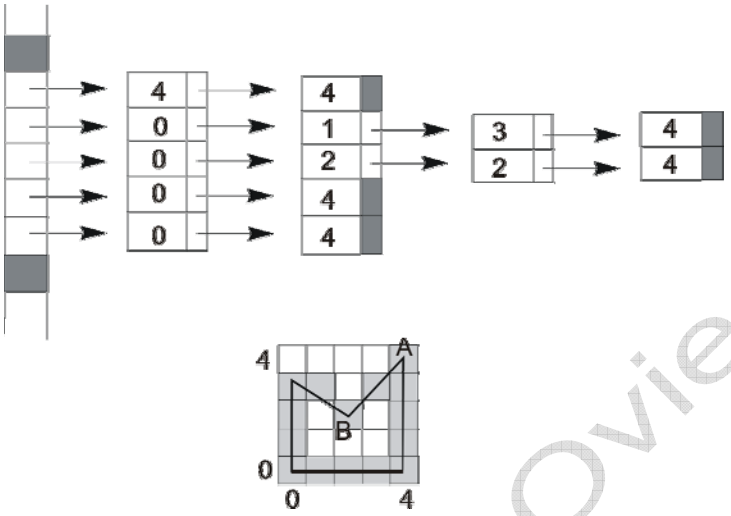


figura 3: vector de listas enlazadas para registrar los segmentos raster

Como se puede ver, a cada línea raster del monitor se le asocia una lista encadenada, *en la que se insertan nodos (uno por cada extremo de los segmentos raster) con la información necesaria para discretizar los segmentos que haya en la línea scan.* Por tanto, si en una línea scan hay "*r*" segmentos, el total de nodos en la lista correspondiente será de "2 *r*".

Según el apartado anterior, el formato mínimo de los nodos queda como sigue:

X_{SRPV}	I_i o $\mathbf{N}_v$	Z_{SRV}

donde

X_{SRPV}: coordenada X de inicio o final de un segmento raster
I_i: intensidad en uno de los extremos de un segmento raster
$\mathbf{N}_v$: normal media en uno de los extremos de un segmento raster
Z_{SRV}: profundidad al inicio o final de un segmento raster

Así, tomando un par de nodos correspondientes a un segmento, X_{SRPV} indicaría dónde comienza y finaliza el trazado del segmento raster; I_i, nos dirá cuál es la intensidad inicial y final en del segmento. En el caso de que se utilice N_v, sabremos cuáles son las normales en los extremos del segmento.

Cada polígono que se discretice ha de tener su propio *vector de listas enlazadas*, conocido también como *lista de aristas*.

Este planteamiento, aunque general, conlleva varios problemas cuya solución reduce la velocidad de discretización de los polígonos.

En primer lugar, las coordenadas de los extremos de los segmentos ras-

ter pueden ser calculadas en un orden cualquiera, por lo que *es preciso insertar los nodos ordenadamente*, o bien, ordenar las listas encadenadas al finalizar su creación.

Por otro lado, vemos que los vértices siempre generan un par de nodos. Sin embargo, los nodos generados son diferentes si el vértice es convexo (vértice A, en la figura 3) o cóncavo (vértice B). En el primer caso, un vértice es a la vez, el inicio y el final de un segmento raster, mientras que en los vértices cóncavos (B), es el final de un segmento y el comienzo de otro. Para detectar de qué tipo es el vértice en cuestión, es preciso llevar un control sobre la pendiente de las aristas, es decir, controlar si son aristas crecientes o decrecientes, con respecto al sentido de discretización. Esto supone un tiempo extra de renderizado.

Estos problemas se eliminan de un plumazo si:

a) *los modelos son estrictamente eulerianos*, es decir, no se permiten agujeros en los polígonos, y si

b) *sólo hay polígonos convexos.*

La primera restricción no es un gran problema, ya que, según vimos en su momento, los esquemas de modelado sólido trabajan normalmente con objetos de Euler.

En cuanto a la obligación de operar exclusivamente con polígonos convexos, tampoco supone una limitación al modelado, ya que *cualquier polígono cóncavo puede descomponerse en dos o más polígonos convexos*. De hecho, los principales algoritmos de teselación generan solamente polígonos convexos.

Por definición, *un polígono es convexo si no existe una recta que corte a sus aristas en más de dos puntos*. Por lo tanto, ninguna línea raster cortará en más de dos puntos a los polígonos convexos, lo que implica que:

a) la lista de aristas se transforma en una simple matriz nodos (*matriz de aristas*), lo que ahorra espacio y aumenta la velocidad de acceso y gestión

b) la ordenación se simplifica hasta el extremo de tener que ordenar sólo dos nodos, por cada línea raster

c) desaparece el problema de los vértices, dado que todos son convexos.

Cuando se estudien los procesos que colorean las superficies de los polígonos, veremos que existen más razones para tratar de evitar los polígonos cóncavos.

Veamos seguidamente cómo calculan los *procesos de discretización* los valores $X_{SRPV,}$ I_i o $\mathbf{N_v}$ que se registran en los nodos de las listas.

C) Procesos de discretización

i.- Localización de los segmentos raster

Encontrar los segmentos de rasterización es lo mismo que *localizar el contorno o puntos de frontera del polígono*. Según hemos visto, en cada nodo se ha de registrar la coordenada X (X_{SRPV}) de inicio o final del segmento de rasterización. Ambas coordenadas no son otra cosa que los *puntos de intersección de las aristas del polígono con la línea scan*. Por lo tanto, como las *aristas son rectas*, para encontrar los extremos de los segmentos de rasterización, *lo más efectivo es trazar las aristas mediante un algoritmo trazador de líneas*, ya que pueden llegar a ser muy rápidos.

Imaginando los píxeles (puntos del espacio discreto) como una red uniforme de círculos, en la figura 4 puede verse un ejemplo de trazado de una línea. Los píxeles pintados, que normalmente son los más cercanos a la línea teórica (continua), aparecen como círculos negros.

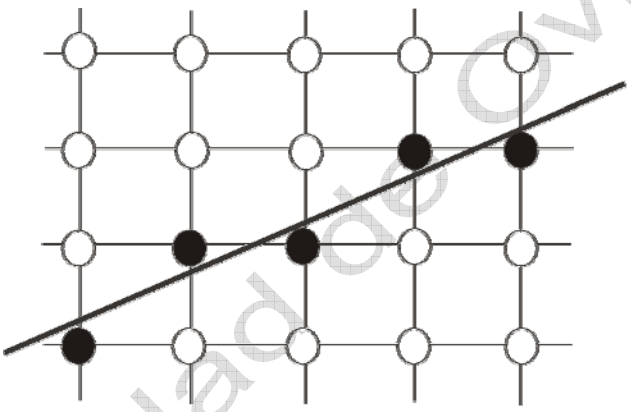


figura 4: trazado de una línea discreta, mostrando los píxeles pintados

Dada la naturaleza discreta del espacio raster, los píxeles se direccionan mediante coordenadas enteras. Por tanto, la recta trazada (discreta) sólo podrá aproximarse a los puntos de la línea teórica (continua). Por ejemplo, el punto (10.33, 20.72) de la línea continua se discretiza en la posición o píxel (10, 21). Este redondeo de las coordenadas a valores enteros ocasiona una pérdida de información (aliasing), dando lugar a una apariencia típica de líneas escalonadas.

Algoritmos trazadores de líneas

En la visualización standard puede utilizarse cualquiera de los algoritmos de rasterización de aristas, aunque, como era de esperar, unos son más apropiados que otros.

Por ejemplo, se pueden rasterizar las aristas a partir de la *ecuación de la recta* que las define. Este sería uno de los métodos más sencillos, pero también uno de los más costosos en tiempo de cálculo. Otra posibilidad sería aplicar algoritmos como el de *Bresenham*, o bien el del *punto medio*, aunque éstos son más apropiados para los casos en los que no se efectúa la rasterización de las superficies, es decir, cuando solamente se han de trazar las aristas. La razón está en que ambos algoritmos son muy meticulosos en la

búsqueda de los píxeles apropiados, lo que supone un tiempo extra de cálculo.

Un algoritmo menos perfeccionista en la búsqueda de los píxeles es el **DDA** (**A**nalizador **D**iferencial **D**igital). Su principal defecto es que no se esmera mucho en localizar los píxeles más apropiados, y además, *de vez en cuando se le escapa alguno* (dependiendo de la versión utilizada), como puede verse en la figura 5-a. Por contra, su mayor virtud es la velocidad de trazado.

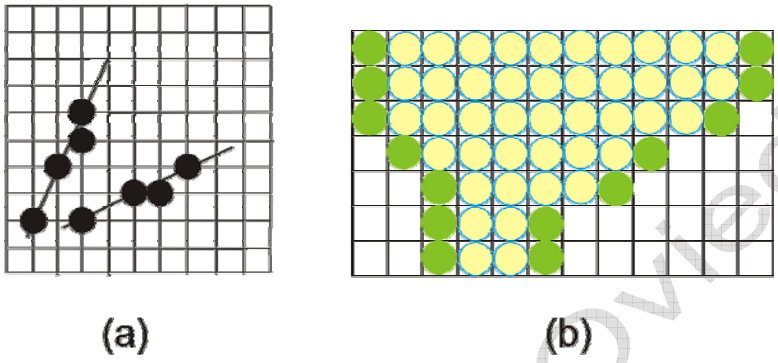


figura 5: huecos dejados por el DDA

Estos lapsus del DDA no representan un gran problema cuando además de discretizar las aristas, también se rasterizan las superficies, ya que entonces las imperfecciones visuales que causan pasan inadvertidas (figura 5-b). Como este es el caso en el rendering standard, veamos con más detenimiento dicho algoritmo.

Algoritmo DDA

La idea básica del DDA consiste en calcular cuánto incrementa la coordenada x (Δx) por cada línea de rastreo ($\Delta y = 1$). Los incrementos serán positivos o negativos, dependiendo de las posiciones de los extremos, y del sentido de rasterización.

Siendo (xi, yi) y (xf, yf) las coordenadas enteras correspondientes a los pixeles inicial y final de la recta, y suponiendo que yf > yi, la versión más simple del DDA quedaría como sigue:

```
DDA_reales(int xi, int yi, int xf, int yf)
{
   int y;
   float x, delta_x;
   x = xi; //Valor inicial de x

   delta_x = (xf - xi)/(yf - yi); //Incremento en X.
   for(y = yi; y < yf; y++)
   {
      SetPixel1(redondear(x),y); //Se pinta el píxel
      x += delta_x; //Se incrementan x
   }
}/* fin DDA_reales() */
```

algoritmo 1: DDA con aritmética decimal

Ver que el DDA selecciona los píxeles cuya distancia a la línea real sea menor, como puede verse en la figura 6.

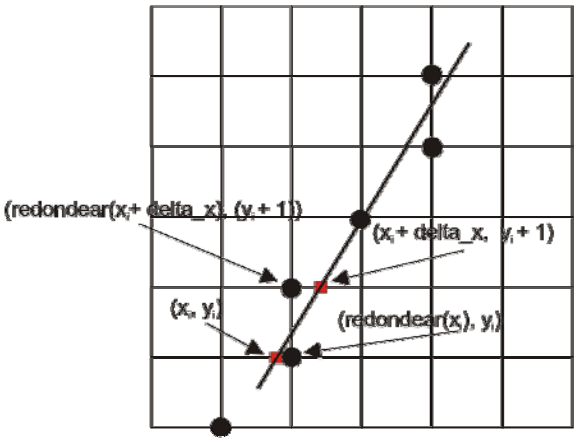


figura 6: el DDA selecciona los píxeles más próximos a la línea teórica

(xf – xi) y (yf – yi) son las distancias entre los extremos de la recta, en el eje X y en el Y, respectivamente. Si (xf – xi) > (yf – yi), entonces

$$\Delta x = (xf - xi) / (yf - yi) > 1$$

Cuando esto ocurre, es cuando el DDA puede perder (dejar sin pintar) alguno de los píxeles en el eje X.

Si se desea que esto no suceda, hemos de utilizar una versión del DDA que seleccione como eje de rasterización (incremento 1), aquél cuya distancia recorrida sea la mayor. El algoritmo 2 muestra dicha versión.

```
DDA_reales(int xi, int yi, int xf, int yf)
{
   int dx, dy; //Distancia en X e Y
   int distancia_max; //Contador auxiliar
   float delta_x, delta_y;
   float x, y; //Coordenadas reales en X e Y

   dx = xf - xi; //Distancia en X (valores enteros)
   dy = yf - yi; //Distancia en Y (valores enteros)

   if(abs(dx) > abs(dy)) //Se determina qué eje tendrá inc. 1
    distancia_max = abs(dx);
   else
    distancia_max = abs(dy);

   delta_x = dx/distancia_max; //Se calculan los incrementos.
   delta_y = dy/distancia_max; //Uno de los dos ha de ser 1.0

   x = xi; //Se inicializan las coordenadas reales
   y = yi;

   while(distancia_max)
   {
      SetPixel(redondear(x),redondear(y));
      x += delta_x;
      y += delta_y;
      distancia_max--;
   }
}/* fin DDA_reales() */
```

algoritmo 2: DDA con aritmética decimal, sin pérdida de píxeles

Vemos un ejemplo del algoritmo anterior.

dx = 5
dy = 3
distancia_max = 5
delta_x = 1.0
delta_y = 0.6
(xi, yi) = (1, 1)
(xf, yf) = (6, 4)
x = 1.0
y = 1.0

x	y	distancia_max	redondear(x)	redondear(y)
1.0	1.0	5	1	1
2.0	1.4	4	2	1
3.0	2.2	3	3	2
4.0	2.8	2	4	3
5.0	3.4	1	5	3
6.0	4.0	0	6	4

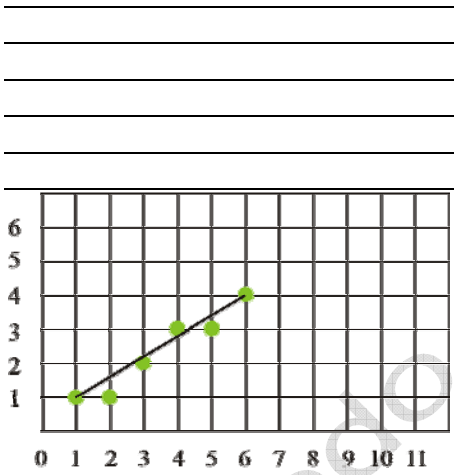


figura 7: ejemplo de rasterización mediante el DDA

En el caso del rendering standard, suele preferirse una versión del DDA similar a la mostrada en el algoritmo 1, dado que, al no importar que se despiste algún píxel, es más rápida y fácil de implantar en VLSI. El principal inconveniente de esta versión es que ha de trabajar en coma flotante, lo que hace necesario un redondeo en cada iteración para calcular las coordenadas enteras de los píxeles, y por tanto ralentiza el algoritmo.

Existen versiones mejoradas de este algoritmo, que trabajan exclusivamente con enteros. Veamos una.

Si llamamos:

```
xe, a la parte entera de x
xd, a la parte decimal de x
delta_xe, a la parte entera del incremento en X
delta_xd, a la parte decimal del incremento en X
```

entonces, el algoritmo 1 puede reescribirse de la siguiente manera:

```
DDA_mixto(int xi, int yi, int xf, int yf)
{
  int xe, delta_xe, y;
  float xd, delta_xd;
  xe = xi; //Valor inicial de x
  xd = -0.5;

  delta_xe = (xf - xi) / (yf - yi); //Incremento entero en X
  //Inc.real en X
  delta_xd = (float)(xf - xi) / (yf - yi) - delta_xe;

  for(y = yi; y < yf; y++)
  {
    SetPixel(xe, y); //Se pinta el píxel
    xe += delta_xe; //Se incrementan la parte entera de x
    xd += delta_xd; //Se incrementan la parte real de x
    if (xd >= 0.0) //Se verifica si se desborda la aparte real
    {
      xe++; //Se incrementa la parte entera
      xd -= 1.0; //Se vacía el control de overflow
    }
```

```
    }
} /* fin DDA_mixto() */
```

algoritmo 3: DDA con aritmética mixta (enteros y reales)

"xd" se inicializa a -0.5 para centrarse en el píxel, hacer el redondeo más fácil y simplificar la condición de overflow.

En el algoritmo anterior aún se trabaja con valores en coma flotante, por lo que no representa ninguna mejora con respecto a la versión inicial. Sin embargo, vemos que *la parte decimal sólo se utiliza para controlar el desbordamiento de las cifras decimales*, y sumar 1 a la parte entera cuando esto ocurre. En esta versión, el control de overflow se realiza en el intervalo real [-1.0, 0.0], pero éste *sería igual de efectivo si se efectuarse en el intervalo entero [-k, 0].*

Por lo tanto, si se desea que el algoritmo 3 trabaje exclusivamente con enteros, lo único que se ha de hacer es *escalar todos los valores decimales del algoritmo multiplicando por k,* de forma que el acumulado de overflow varíe en el intervalo [-*k*, 0].

Según veremos a continuación, un valor apropiado para *k* viene dado por:

ec. 1

$$k = 2(yf - yi)$$

En la versión mixta vemos que:

ec. 2

```
delta_xd = (xf - xi)/(yf - yi) - delta_xe;
```

Recordando que el ***D****ividendo* = ***d****ivisor* · ***c****ociente* + ***r****esto* ($D = d \cdot c + r$), se obtiene la expresión:

ec. 3

$$\frac{D}{d} - c = \frac{r}{d}$$

si se divide por *"d"*, y se resta *"c"*.

Comparando la estructura de las ecuaciones ec. 2 y ec. 3, vemos que en la primera:

ec. 4

```
(xf - xi) ≡ D
(yf - yi) ≡ d
delta_xe ≡ c
```

Por lo tanto, sustituyendo estas equivalencias en ec. 2 queda que:

ec. 5

$$\texttt{delta\_xd} = \frac{D}{d} - c = \frac{r}{d}$$

Recordemos que *r* es el resto, y que por lo tanto puede expresarse como:

$$r = D \bmod d$$

Si también se sustituye "d" en la ec. 1, se puede expresar el factor de escala *k* como:

ec. 6

$$k = 2(yf - yi) = 2d$$

El paralelismo que acabamos de exponer entre la ec. 2 y la ec. 3 *solamente sirve para entender mejor cómo se efectúa el escalado.*

Para escalar el algoritmo 3 de forma que el control de overflow varíe dentro de un rango de valores enteros [-*k*, 0], *se ha de multiplicar cada constante y variable real por el factor de escala "k"*. Siguiendo el orden de aparición en el algoritmo tenemos:

```
1) int xd, delta_xd;
2) xd =[3] -0.5·k; → xd = -1/2·2(yf - yi) = -(yf - yi)
3) delta_xd = (r/d)·k → delta_xd =(r/d)·2d = 2r = 2(D mod d)
           = 2·[(xf - xi) mod (yf - yi)]
4) if(xd >= 0.0·k) → if(xd >= 0)
5) xd -= 1.0·k → xd -= 2(yf - yi)
```

Sustituyendo estas 5 modificaciones, tendremos una versión del DDA trabajando exclusivamente con enteros.

```
DDA_enteros(int xi, int yi, int xf, int yf)
{
   int xe, delta_xe, y;
   int xd, delta_xd; ← 1)
   xe = xi; //Valor inicial de x
   xd = -(yf - yi); ← 2)

   delta_xe = (xf - xi) div (yf - yi); //Incremento entero en X
   delta_xd = 2·[(xf - xi) mod (yf - yi)] ← 3)
   for(y = yi; y < yf; y++)
   {
      SetPixel(xe, y); //Se pinta el píxel
      xe += delta_xe; //Se incrementan la parte entera de x
      xd += delta_xd; //Se incrementan la parte real de x
      if (xd >= 0) ← 4)
      {
         xe++; //Se incrementa la parte entera
         xd -= 2(yf - yi); ← 5)
      }
   }
} /* fin DDA_enteros() */
```

algoritmo 4: DDA con aritmética entera

Ver que *k* es constante en todo el algoritmo, por lo que se puede calcu-

[3] Ver que sólo se multiplica la parte operativa (2º miembro), ya que en realidad no se trata de una ecuación, sino de la forma de expresar en C la operación xd ← –0.5

lar una sola vez, fuera del bucle. Además, la operación "mod" no es necesario realizarla, dado que es el resto de la operación "div".

ii.- Cálculo de la intensidad en los píxeles

Según el orden de ejecución de los procesos de discretización, lo lógico sería ver a continuación cómo se determina la visibilidad de los píxeles. Sin embargo, por comodidad, vamos a tratar primero el modo de calcular la intensidad (color) de los píxeles en los segmentos raster, y por lo tanto en los polígonos.

Para el cálculo de la intensidad de los píxeles, en algún momento se ha de utilizar un modelo de intensidad (normalmente el de Phong), aunque lo que ahora nos interesa no es cómo calcular eficientemente la intensidad en un punto, sino *en todos los puntos del polígono*. Para ello, se utilizan las *técnicas de aplicación de intensidad,* conocidas también como *técnicas incrementales de intensidad*.

Existen dos estrategias a la hora de aplicar intensidad a los polígonos:

a) dar intensidad a todos los puntos (píxeles) de un polígono, *sin tener en cuenta las peculiaridades de los polígonos vecinos*, y

b) dar intensidad al polígono, *teniendo en cuenta a los polígonos vecinos*.

Los procesos de aplicación de intensidad que no tienen en cuenta a los polígonos vecinos se conocen como *métodos de intensidad sobre polígonos;* los que sí tienen en consideración a los polígonos vecinos, forman el grupo de los *métodos de intensidad sobre redes poligonales.*

El acordarse o no de los vecinos a la hora de pintar la superficie de un polígono tiene importantes consecuencias sobre la calidad final de la imagen renderizada. Uno de los principales problemas que surgen al dar intensidad a los poliedros es el de *la visibilidad de las aristas, en las superficies curvas aproximadas mediante polígonos*. Este problema puede llegar a eliminarse, si en la aproximación poligonal se utiliza un número suficiente de polígonos, y si se aplica la intensidad sobre una malla de polígonos, en vez de pintar cada polígono de forma independiente (sin tener presente a los vecinos).

Además de intentar conseguir una calidad visual aceptable, uno de los objetivos primordiales de las técnicas de aplicación de intensidad es el de colorear los polígonos en el menor tiempo posible. El modelo de intensidad de Phong puede requerir más del 50% del tiempo de rendering, por lo que dar intensidad eficientemente es de suma importancia.

Métodos de intensidad sobre polígonos

A este grupo pertenecen los métodos de aplicación intensidad que dan color a los polígonos, sin considerar las influencias de los polígonos colindantes.

Método de intensidad constante

Es el más simple y rápido de los conocidos. También se le conoce como *méto-*

do de intensidad de caras o de planos.

Mediante un modelo de intensidad, *se determina el valor de la intensidad en un punto determinado y se aplica esa intensidad a todo el polígono.*

Esta aproximación es muy limitada y sólo da buenos resultados si se asume que:

- La fuente de luz se encuentra en el infinito, por lo que **N**·**L** es constante en toda la superficie del polígono.
- El observador está también en el infinito, y por tanto **N**·**V** es, igualmente, constante toda la superficie.
- El polígono representa la superficie real del objeto que se modela. No es una aproximación a una superficie curva.

Teniendo presente la última condición, este método es aceptable para la visualización de poliedros que "modelen poliedros", o algo similar. No es recomendable para el renderizado de superficies curvas, aproximadas mediante polígonos, ya que la visibilidad de las aristas sería muy notoria.

Si falla alguna de las 2 primeras hipótesis, entonces para aplicar este modelo debemos buscar un método que determine valores simples de **L** y de **V** para cada polígono. Por ejemplo, los valores pueden calcularse en el centro del polígono o en su primer vértice y aplicarlos a todo el polígono.

Aún cuando se den todas las condiciones anteriores, al ser la intensidad constante, no se producen variaciones de intensidad en todo el polígono, lo que causa un efecto visual poco realista. Sin embargo, como es el método más rápido, puede resultar interesante en la previsualización de los modelos.

Método de interpolación lineal

Para evitar que la intensidad sea constante a lo largo de todo el polígono, el *método de interpolación lineal* propone que la intensidad de cada punto del polígono se calcule mediante interpolación lineal.

Para ello, se toman como valores iniciales las intensidades en los vértices del polígono. Si éste es de n vértices, habría que aplicar n veces el modelo de intensidad, y luego, a partir de la intensidades en estos n puntos, interpolar las intensidades restantes hasta rellenar completamente el polígono.

Este método, aunque supera el resultado visual del anterior, también ha de someterse a las tres condiciones anteriores, si se desean resultados aceptables.

Comparado con el método de intensidad constante, este método es bastante más lento y sus resultados visuales no suponen una mejora radical, por lo que no es aconsejable. La interpolación lineal tiene mejores perspectivas cuando se tiene en consideración a los polígonos del vecindario.

Métodos de intensidad sobre redes poligonales

Según lo comentado arriba, cuando se aproxima una superficie curva mediante una red de polígonos, si cada polígono en la red es pintado individualmente, resulta fácil distinguirlos de sus vecinos, debido al problema de visibilidad de las aristas. Como acabamos de ver, este problema es más notorio si los polígonos son coloreados aplicando el método intensidad constante, el de interpolación lineal o incluso si se calcula la intensidad píxel a píxel. La razón está en que *los polígonos adyacentes, con diferente orientación, tienen intensidades distintas a lo largo de sus bordes*.

La visibilidad de las aristas es acentuada por un efecto visual conocido como *Bandas de Mach* (efecto **Mach-Band**), debido a la inhibición lateral de los ojos. Este efecto exagera el contraste cuando se producen gradientes de intensidad. Estas diferencias de intensidad normalmente ocurren en las frontera entre polígonos con diferente orientación, por lo que en los bordes de los polígonos, la cara oscura parecerá más oscura de lo que es, y la cara clara, más clara.

La solución más simple para evitar o minimizar la visibilidad de las aristas es aproximar las superficies curvas mediante mallas poligonales finas, ya que entonces la orientación entre caras adyacentes varía poco, y por lo tanto se minimizan los gradientes de intensidad. Sin embargo, *esta solución no es suficiente, si no se utiliza un método de aplicación de la intensidad que tenga en cuenta a los polígonos vecinos*, como el de Gouraud o el de Phong, que veremos a continuación.

Método de interpolación de Gouraud

Este método, desarrollado en 1971, fue cronológicamente el primero en superar las limitaciones del método de intensidad constante. Anteriormente, otros autores habían desarrollado algo parecido para los triángulos, pero fue Gouraud quien generalizó esta técnica para polígonos arbitrarios.

El método de interpolación de Gouraud se desarrolla en tres fases:

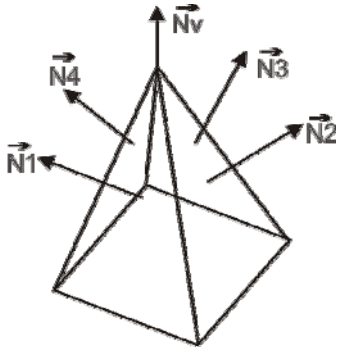
1) Se calcula la intensidad en los vértices del polígono.
2) Partiendo de las intensidades anteriores, se calculan las intensidades en las aristas interpolando.
3) Por último, tomando las intensidades en las aristas como valores iniciales, y siguiendo las líneas scan, se calculan las intensidades de los puntos de interior, también interpolando.

Cálculo de la intensidad en los vértices

Lo que diferencia al método de interpolación de Gouraud del método de interpolación lineal visto arriba, es el modo de calcular los valores iniciales de las intensidades en los vértices.

En este caso, la normal (orientación) que se utiliza no es la del polígono, sino la normal a la superficie algebraica (teórica). Dicha normal se almacena en la base de datos del modelo. Si no está disponible, se calcula la **normal media de los vértices**, que es igual a la *media aritmética de las*

normales de los polígonos que convergen en cada vértice (figura 8). La normal media es una aproximación a la normal teórica.



$$\vec{N}_v = \frac{\sum_i \vec{N}_i}{\sum_i \left|\vec{N}_i\right|}$$

figura 8: cálculo de la normal media

Una vez que se dispone de la normal real o de la normal media de un vértice, se utiliza dicho vector en el modelo de intensidad (generalmente el de Phong), para calcular la intensidad en el vértice.

Algunos sistemas, a la hora de calcular la intensidad en los vértices, *eliminan la componente especular* del modelo de intensidad, debido a que, con Gouraud, la forma e intensidad de los brillos especulares dependen mucho de las características geométricas de la red poligonal, lo que genera resultados imprevistos.

Por otro lado, para que la intensidad calculada dependa exclusivamente de la normal en los vértices, también suele suponerse que las fuentes y el observador se encuentran en el infinito.

Cálculo de las intensidades en las aristas

Suponiendo que ya se dispone de las intensidades en los vértices de un polígono (I_1, I_2 e I_3, en la figura 9), el siguiente paso en el método de interpolación de Gouraud consiste en calcular las intensidades en las aristas. Veamos cómo se efectúa el cálculo de la intensidad en dos aristas diferentes, en los puntos I_a e I_b, respectivamente.

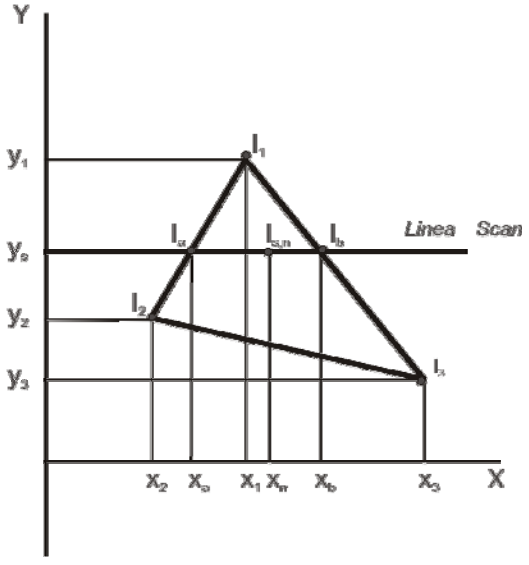


figura 9: extremos de interpolación en el método de Gouraud

Para I_a queda:

ec. 7

$$I_a = \frac{1}{y_1 - y_2}\left[I_1\left(y_s - y_2\right) + I_2\left(y_1 - y_s\right)\right] \rightarrow I_a = I_1 - \left(I_1 - I_2\right)\frac{\left(y_1 - y_s\right)}{\left(y_1 - y_2\right)}$$

Si a la ecuación de la izquierda se le agrega el término $\left(I_1 y_1 - I_1 y_1\right)$, la ecuación no varía. Reagrupando, se obtiene la ecuación de la derecha, que es más eficiente, pues requiere un producto menos.

Ver que cuando ($y_s - y_2$) $\cong 0 \Rightarrow I_a \cong I_2$, ya que el punto de la arista se encuentra cerca del vértice (x_2, y_2). Si ($y_1 - y_s$) $\cong 0$ entonces $I_a \cong I_1$, ya que en este caso nos encontramos en las cercanías del vértice (x_1, y_1). En definitiva, las intensidades a lo largo de la arista varían entre I_1 e I_2. Ver que se divide por la distancia máxima (en Y) entre los vértices, o sea por ($y_1 - y_2$), para normalizar el resultado.

En la otra arista queda:

ec. 8

$$I_b = \frac{1}{y_1 - y_3}\left[I_1\left(y_s - y_3\right) + I_3\left(y_1 - y_s\right)\right] \rightarrow I_b = I_1 - \left(I_1 - I_3\right)\frac{\left(y_1 - y_s\right)}{\left(y_1 - y_3\right)}$$

El análisis y conclusiones son similares a los de la ecuación anterior.

Cálculo de las intensidades en la superficie

En la tercera y última fase, el método de interpolación de Gouraud calcula las intensidades de los puntos de interior, a partir de los valores calculados en las aristas. La interpolación se efectúa siguiendo la línea scan, y por tanto, las intensidades utilizadas en la interpolación son las que haya en los puntos de intersección de las aristas con la línea scan (I_a e I_b, en la figura 9).

La ecuación de interpolación para calcular la intensidad en los puntos de interior es similar a las anteriores. La intensidad $I_{s,n}$ en un punto de la superficie viene dada por:

ec. 9

$$I_{s,n} = \frac{1}{x_b - x_a}\left[I_a\left(x_b - y_{s,n}\right) + I_b\left(x_{s,n} - x_a\right)\right] \rightarrow I_{s,n} = I_b - \left(I_b - I_a\right)\frac{\left(x_b - x_{s,n}\right)}{\left(x_b - x_a\right)}$$

Para ser más eficientes, estas ecuaciones, son implementadas usando cálculos incrementales. Esto es particularmente importante en esta última ecuación, ya que normalmente los puntos de interior abundan mucho más que los de frontera.

Tomando de nuevo la figura 9 como referencia, la intensidad en $I_{s,n}$ y en $I_{s,n+1}$ está dada por:

ec. 10

$$I_{s,n} = I_b - \left(I_b - I_a\right)\frac{\left(x_b - x_{s,n}\right)}{\left(x_b - x_a\right)}$$

$$I_{s,n+1} = I_b - \left(I_b - I_a\right)\frac{\left(x_b - x_{s,n+1}\right)}{\left(x_b - x_a\right)}$$

Siendo $\Delta I_s = I_{s,n+1} - I_{s,n}$, sustituyendo las ecuaciones anteriores, operando y simplificando se llega a:

ec. 11

$$\Delta I_s = \frac{\Delta x}{\left(x_b - x_a\right)} \cdot \left(I_b - I_a\right)$$

donde $\Delta x = (x_{s,n+1} - x_{s,n})$, es decir, la distancia incremental a lo largo de la línea scan.

Según esto, la intensidad en el píxel n de la línea scan s se calcula como:

ec. 12

$$I_{s,n} = I_{s,n-1} + \Delta I_s$$

De modo análogo se deducen las ecuaciones incrementales para las aristas.

Después de ver cómo se calcula la intensidad en la superficie, podemos entender mejor por qué el método de interpolación de Gouraud puede tener problemas con el cálculo de la componente especular.

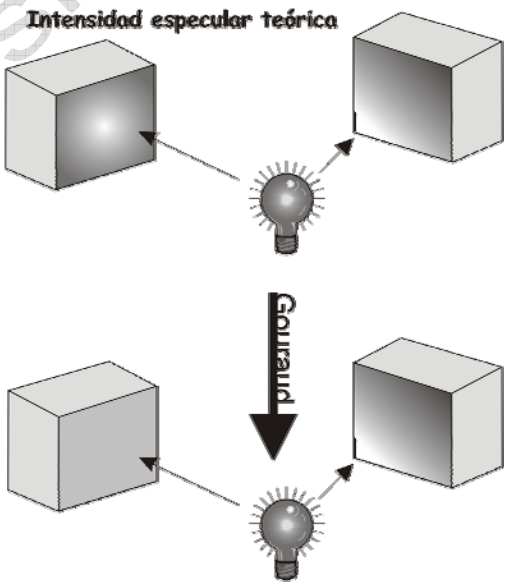


figura 10: pérdida de la componente especular al aplicar Gouraud

Como la intensidad en todo el polígono se calcula a partir de la intensidad en los vértices, si coincide que en éstos no hay reflejo especular, tampoco

puede haberlo en el resto del polígono. Por lo tanto, un reflejo especular que coincida en el centro del polígono (sin tocar los vértices), no será tenido en cuenta (figura 10).

Método de interpolación de Phong

El modo de proceder del método de Phong es muy similar al de Gouraud. Sin embargo, en vez de interpolar las intensidades, *interpola las normales en los vértices*. Si no se dispone de las normales a las superficies teóricas, se calculan de la misma forma que para el método de Gouraud.

Las normales en los vértices se interpolan, a su vez, para calcular las normales en los puntos de las aristas (extremos de los segmentos raster) y posteriormente se repite el proceso para cada punto (píxel) de interior. Para cada línea scan, *la normal interpolada debe ser normalizada*. En la siguiente figura se muestran dos normales y sus interpolaciones antes y después de ser normalizadas.

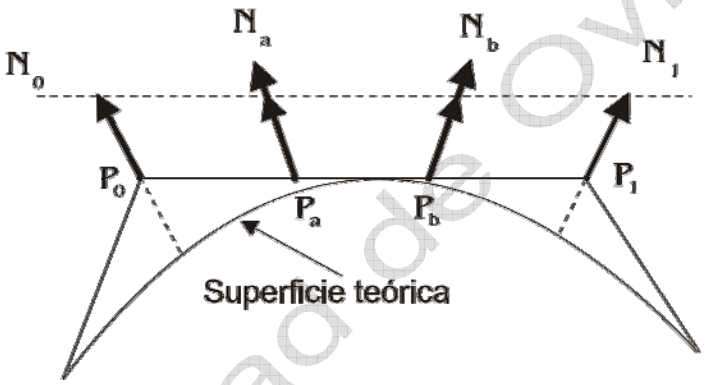


figura 11: interpolación de las normales, y su normalización

Ver que cada normal interpolada en los polígonos, *es una aproximación a la normal real de la superficie teórica*. Este hecho es el que proporciona calidad al método de interpolación de Phong, ya que las siluetas poligonales se disimulan al máximo.

Ecuaciones de interpolación

Salvo por el hecho de que trabaja con normales, la mecánica de interpolación en el método de Phong es igual a la de Gouraud. Las ecuaciones, basadas en este caso en la figura 12, quedan como:

ec. 13

$$N_a = \frac{1}{y_1 - y_2}[N_1(y_s - y_2) + N_2(y_1 - y_s)]$$

$$N_b = \frac{1}{y_1 - y_4}[N_1(y_s - y_4) + N_4(y_1 - y_s)]$$

$$N_s = \frac{1}{x_b - x_a}[N_a(x_b - x_s) + N_b(x_s - x_a)]$$

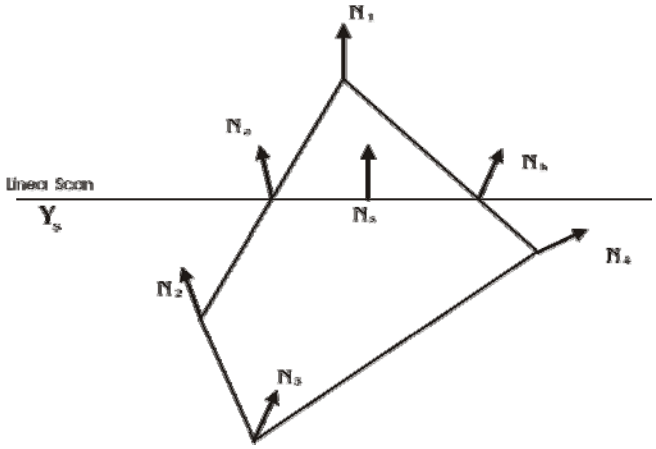


figura 12: método de interpolación de Phong

Estas ecuaciones, al igual que en Gouraud, se pueden expresar en formato incremental. Si definimos Δx como la distancia incremental a lo largo de la línea scan, el vector de incrementos $\Delta\mathbf{N_s} = (\Delta\mathbf{N_{sx}}, \Delta\mathbf{N_{sy}}, \Delta\mathbf{N_{sz}})$ viene dado por:

ec. 14

$$\Delta N_{sx} = \frac{\Delta x}{x_b - x_a}(N_{bx} - N_{ax})$$

$$\Delta N_{sy} = \frac{\Delta x}{x_b - x_a}(N_{by} - N_{ay})$$

$$\Delta N_{sz} = \frac{\Delta x}{x_b - x_a}(N_{bz} - N_{az})$$

Aplicando estos incrementos, el cálculo de cada componente de la normal $\mathbf{N_s}$ queda como sigue:

ec. 15

$$N_{sx,n} = N_{sx,n-1} + \Delta N_{sx}$$

$$N_{sy,n} = N_{sy,n-1} + \Delta N_{sy}$$

$$N_{sz,n} = N_{sz,n-1} + \Delta N_{sz}$$

Las ecuaciones anteriores se deducen de forma similar a como vimos en Gouraud.

Cálculo de la intensidad

El vector normal interpolado se utiliza finalmente en el cálculo de la intensidad. Por lo tanto, *para cada píxel del polígono se ha de aplicar el modelo de intensidad*.

Al igual que en Gouraud, para facilitar los cálculos en el modelo de intensidad, *se supone que la fuente de luz y el observador se encuentran en el infinito*. De esta forma, la intensidad en un punto es sólo función de la normal.

Aceleración del método de Phong

Acabamos de ver que el método de Phong requiere la aplicación del modelo de intensidad en cada punto del polígono, mientras que en Gouraud solamente se necesita en los vértices. Además, Phong interpola vectores, lo que hace que, sólo en este aspecto, sea tres veces más lento que el método de Gouraud.

En definitiva, el método de interpolación de Phong es bastante más lento que el de Gouraud, lo cual puede llegar a ser un problema en algunas aplicaciones gráficas, p. ej. en la producción de secuencias animadas. Soluciones para acelerar el método de interpolación Phong no faltan.

Una de ellas consiste en interpolar la normal cada dos puntos, en lugar de hacerlo punto por punto. Así, si se interpolan las normales en los píxeles 1, 3, 5, 7..., el píxel número 2 se calcula como la media del píxel 1 y 3, el 4 como la media del 3 y el 5, etc. La interpolación alternante apenas supone pérdida de calidad en la imagen.

También se puede combinar Phong y Gouraud, dependiendo del resultado del **Test H** o *test de Harrison*. Este test detecta qué polígonos tienen total o parcialmente, áreas de luz especular. Se basa en un conjunto de pruebas simples que predicen el valor del brillo sobre una línea entre dos vértices.

Según se ha comentado arriba, el método de Gouraud puede tener problemas con el cálculo correcto de la componente especular. Por fortuna, los polígonos iluminados especularmente suelen ser relativamente pocos. Por tanto, resulta más eficiente aplicar Phong en los polígonos iluminados especularmente y Gouraud los restantes. Sin embargo, la intensidad difusa calculada mediante Phong difiere levemente de la hallada aplicando Gouraud. Por este motivo, en la práctica es preferible utilizar Gouraud para calcular la intensidad difusa en todo el objeto, y aplicar Phong para evaluar la componente especular en los polígonos indicados por el Test H. Posteriormente, la intensidad especular es combinada con el valor de la luz difusa.

Comparación entre el método de Gouraud y el de Phong

- El método de Phong genera una iluminación de las superficies más real que el de Gouraud. Las siluetas poligonales se reducen mucho con respecto a Gouraud, aunque no llegan a desaparecer.
- El efecto Mach-Band, en general, es más pequeño que en Gouraud. Sin embargo, se ha demostrado que en algunos casos, como por ejemplo en las esferas, Phong produce peores efectos Mach-Band que el método de Gouraud.
- El método de Phong, al aplicar el modelo de intensidad a cada píxel del polígono, no da problemas al trabajar con la componente especular. En otras palabras, el método de interpolación de Phong depende mucho menos de las características geométricas de la malla poligonal que el de Gouraud.
- Por contra, el método de Phong requiere muchos más cálculos que el

de Gouraud, ya que además de interpolar vectores, ha de calcular la ecuación de intensidad en cada píxel; con Gouraud sólo se aplica el modelo de intensidad en los vértices, y además interpola valores escalares (intensidades).

Problemas comunes a los métodos de interpolación

A la hora de dar intensidad a los polígonos interpolando, se presentan una serie de problemas comunes a todos los métodos, algunos de los cuales ya han sido comentados. Los más importantes son:

- **Silueta poligonal**. Aproximando una superficie curva mediante polígonos y aplicando el modelo de interpolación, la silueta obtenida es todavía poligonal, a pesar de que al aplicar Phong se tiende a recuperar la curvatura de la superficie original, al calcular los vectores normales interpolados (figura 11). *Se puede reducir la visibilidad de las aristas dividiendo la superficie en un número muy grande de pequeños polígonos*. Esta solución tiene el inconveniente de encarecer la rasterización, y el rendering en general.
- **Distorsión de la perspectiva**. Se producen anomalías en la perspectiva del objeto, ya que no se interpola con la información disponible del SUR, sino con la que se tiene en el SRV y el SRPV después de efectuar las aplicaciones SUR → SRV → SRPV. En la rasterización de las aristas, el proceso de interpolación *suma (o resta) una cantidad constante al cambiar de una línea scan a otra*, lo que puede llegar a representar un reparto poco apropiado de la intensidad, cuando los extremos de la arista están muy separados en el eje Z del SUR.

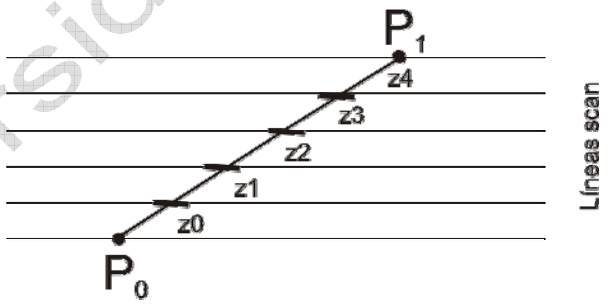


figura 13: distorsión de la perspectiva

Por ejemplo, en la figura anterior, sean $(z0, ..., z4)_{SUR}$ los segmentos de la arista (P_0, P_1) en el SUR, cuyos extremos, proyectados sobre el plano visual, se corresponden con píxeles de las líneas scan. Cuando se interpola entre P_0 y P_1 (en el SRPV), para calcular la intensidad del nuevo píxel al cambiar de línea scan, se suma una cantidad constante, por lo que *implícitamente se está suponiendo que los segmentos* $(z0, ..., z4)_{SUR}$ *son todos iguales*. Sin embargo, si $P_0(Z_{SUR}) << P_1(Z_{SUR})$, dada la naturaleza de la proyección en perspectiva, ocurre que $z0 < z1 < z2 < z3 < z4$. Por tanto, para evitar la distorsión, la intensidad interpolada debería ser proporcional a la longitud de los segmentos en el SUR, en vez de distribuirla equitativamente entre

éstos. *Disminuyendo el tamaño de los polígonos (o lo que es igual aumentando su número), se reduce el tamaño de los segmentos*, impidiendo, de este modo, que existan grandes diferencias de longitud entre ellos.

- **Dependencia de la orientación**. El resultado de la interpolación, no es independiente de la orientación del polígono. Debido a que los polígonos son rasterizados horizontalmente, el resultado puede variar si el polígono es girado. Esto es especialmente grave en animación.

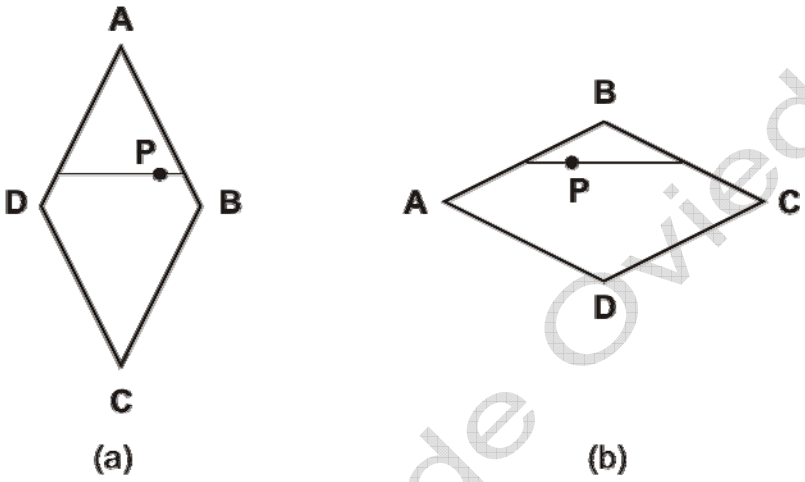


figura 14: dependencia de la orientación del polígono

En la figura anterior, el punto P esta en el mismo polígono pero con distinta orientación. En la figura 14-a, P está interpolado por las intensidades en A, B y D; el mismo punto en el polígono (b) está interpolado por las intensidades en A, B y C. Este problema podría solucionarse descomponiendo el polígono en triángulos, una de las razones que tienen los *algoritmos de teselación* para inclinarse por este tipo de polígonos. Hay algoritmos que lo corrigen, pero son muy caros en tiempo del cálculo.

- **Vértices compartidos**. Se pueden producir problemas cuando dos polígonos adyacentes comparten un vértice. En la figura 15 se observan tres polígonos y un vértice C compartido por dos de ellos. En la interpolación de los polígonos de la derecha, que comparten el vértice C, intervienen los valores en A, C y B, mientras que en el polígono de la izquierda sólo intervienen A y B. Por lo tanto, se producirá una discontinuidad en la intensidad, que el efecto Mach-Band se encargará de resaltar. *Esta discontinuidad se puede eliminar insertando en el polígono de la izquierda un vértice extra que tenga la misma información de interpolación que C.*

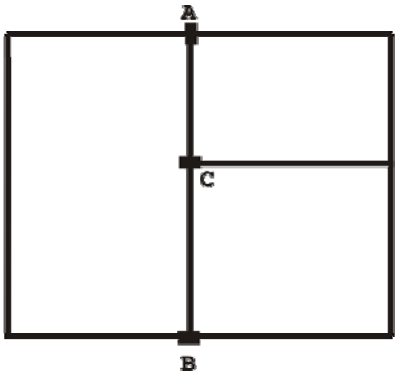


figura 15: vértices compartidos

- **Normales a los vértices no representativas**. Puede que las normales a los vértices no representen adecuadamente la geometría de la superficie del objeto. Al calcular las normales a los vértices como la media de todas las normales a los polígonos que comparten ese vértice, puede ocurrir que los vectores resultantes sean paralelos (figura 16), con lo que no se producirá variación en el sombreado, si la fuente de luz está muy distante. Nuevamente, *la solución pasa por subdividir los polígonos en otros más pequeños*.

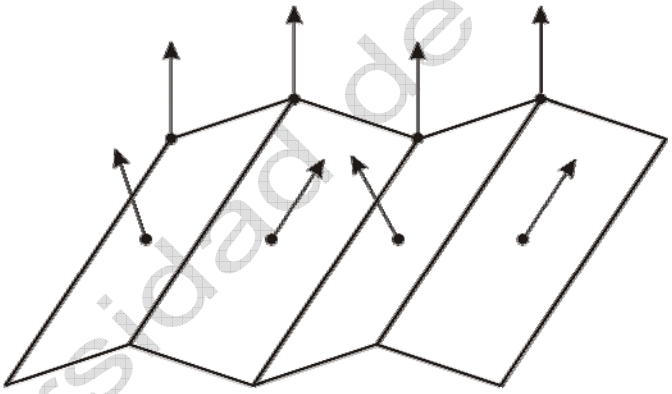


figura 16: normales medias paralelas

- **Problemas con polígonos cóncavos.** La interpolación no funciona correctamente con los polígonos cóncavos. Por ejemplo, en la figura 17, en la línea scan numerada como 1, al calcular la intensidad en cada punto, se utiliza la información proporcionada por los vértices S, Q, y R. Sin embargo, cuando se interpola la línea scan 2, también se utiliza la información proporcionada por el vértice P. Esto puede producir riesgos de discontinuidad en la intensidad.

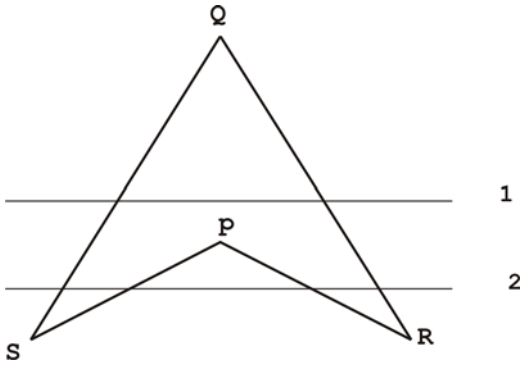


figura 17: rasterización en polígonos cóncavos

Este problema proporciona una razón más para evitar los polígonos cóncavos en el modelado.

iii.- Determinación de la visibilidad de los píxeles

Para finalizar el estudio del proceso de discretización de la imagen vectorial, veamos ahora otro de los problemas típicos del rendering standard, como es el de la eliminación de las superficies y líneas ocultas.

El planteamiento del problema es simple: a partir de las coordenadas de los objetos y del observador en el SUR, averiguar qué puntos son visibles por el observador y cuales no. Existe un amplio abanico de algoritmos para determinar la visibilidad de los polígonos (y superficies en general), catalogados normalmente en función de su velocidad y del espacio donde operan.

A pesar de la diversidad de algoritmos, todos tienen en común que se ha de llevar a cabo algún tipo de ordenación. La más frecuente ordena en función de la distancia entre las superficies y el observador. El quid de esta ordenación está en suponer que, *cuanto más lejos se encuentre un objeto del observador, más fácil será que sea ocultado total o parcialmente por otro objeto más cercano.*

Una vez establecida la distancia al observador (en Z*),* se procede a la ordenación en X e Y para determinar si, en efecto, el objeto es ocultado por otro más próximo al observador. La eficiencia de los algoritmos de eliminación de superficies ocultas depende significativamente de la eficacia del proceso de ordenación.

Los algoritmos de eliminación de líneas y superficies ocultas están clasificados de acuerdo con el espacio donde operan, es decir, si trabajan en el SRPV (*espacio de la imagen*) o directamente en el SRV o SUR (*espacio de los objetos*). En el primer caso, se determina qué polígonos son visibles en cada píxel. Si hay n polígonos*,* por cada píxel se ha de verificar cuál es el más cercano al observador, según indique la línea de proyección a través del píxel. Siendo p el número de píxeles, el trabajo realizado será proporcional a np. En el segundo caso (algoritmos que trabajan en el espacio de los objetos), se comparan los objetos y superficies directamente entre sí, eliminando las superficies (u objetos) que no sean visibles. El trabajo ahora será proporcional a n^2. Aunque lo normal es que $n < p \Rightarrow n^2 < np$, sin embargo, los pasos individuales de este segundo método son mucho más complejos y consumen más

tiempo. Por lo tanto, los algoritmos que trabajan en el SRV generalmente son más lentos y difíciles de implementar.

La mayor o menor velocidad de estos algoritmos no implica que unos sean mejores que otros, ya que, como norma general, *cuanto mayor es su velocidad menores son las prestaciones que ofrecen*, y viceversa. Así, los más rápidos son idóneos para el rendering en tiempo real, mientras que los lentos son apropiados para la animación por ordenador, ya que suelen incorporar transparencias, sombreado, etc., lo que proporciona mayor calidad de imagen.

La idoneidad de unos algoritmos u otros, depende además del tipo de aplicación en que van a ser implantados. En la Síntesis de Imágenes, el algoritmo conocido como *Z-buffer* es, sin duda, el que mayor aceptación ha tenido, gracias a su sencillez, rapidez y generalidad.

Algoritmo Z-buffer

El algoritmo Z-buffer es uno de los más sencillos de implementar, tanto en software como en hardware. Opera en el espacio de la imagen, y fue propuesto originalmente por Latmull (1975).

La idea principal tras el algoritmo Z-buffer es la de utilizar un buffer para almacenar la profundidad de los puntos de los polígonos discretizados. Este buffer, llamado Z-buffer porque almacena coordenadas Z, *posee tantos registros como píxeles hay en la pantalla.*

Inicialmente el Z-buffer se inicializa con el valor máximo, para luego buscar el valor de *z* mínimo, o bien se inicializa a 0 y se busca el máximo.

Suponiendo que se inicializa con el valor máximo, *el algoritmo Z-buffer básico se limita a buscar la profundidad mínima para cada píxel (x, y) del plano discreto.* Así, mientras se discretiza un polígono, se compara la profundidad de cada píxel del polígono, con la coordenada *z* guardada en el registro correspondiente del Z-buffer. Si la comparación indica que el nuevo píxel está por delante del almacenado en el Z-buffer (o sea, más cerca del observador) su coordenada *z* pasará a ser el nuevo valor registrado en el Z-buffer, y el color del píxel se guarda en un buffer de la pantalla (pagina gráfica) para, posiblemente, ser visualizado.

El pseudocódigo del algoritmo podría ser el siguiente:

```
Z-buffer()
{
   //Se inicializan el Z-buffer y el buffer de pantalla
   for (y=0; y<Ymax; i++)
   {
      for (x=0; x<Xmax; x++)
      {
         //Se pone el buffer de pantalla con el color de fondo
         EscribirPixel (x, y, ColorFondo);
         //Se inicializa el Z-buffer con el valor máximo
         Z_buffer(x, y) = Zmax;
      }
   }

   //Comienza la búsqueda de la profundidad mínima
   Para cada polígono
```

```
    {
      Para cada píxel de la proyección del polígono
      {
        //Se averigua la profundidad del píxel
        pz = profundidad(x, y);
        if (pz <= Z_buffer(x, y))
        {
          //Se guarda el nuevo valor el Z-buffer
          Z_buffer(x, y) = pz;
          //Se guarda el color del píxel
          EscribirPixel (x, y, ColorPoligono en píxel(x, y));
        }
      }
    }
  }/* fin Z-buffer */
```

algoritmo 5: Z-buffer

En el pseudocódigo anterior vemos que no existen comparaciones entre objetos y que, en principio, tampoco se requiere ningún tipo de ordenación para obtener resultados satisfactorios. Sin embargo, dado que **no** es necesario calcular la intensidad en los píxeles cuando **no** se cumple la condición de visibilidad, *se puede aumentar considerablemente la eficiencia del algoritmo si se realiza una preordenación de los polígonos, según su distancia al observador, y se comienza la discretización comenzando por los más próximos.*

Cálculo de la profundidad

El método que se utilice para calcular la coordenada Z ha de ser ante todo rápido y conciso. Veamos dos maneras de calcular la profundidad que responden a estos requerimientos.

Método incremental a partir de la ecuación del plano

Los cálculos para la obtención de z en cada píxel de un segmento raster, se simplifican explotando el hecho de que los polígonos son planos.

Partiendo de la ecuación del plano donde se encuentra el polígono

$$Ax + By + Cz + D = 0$$

se despeja la variable z. Queda:

ec. 16

$$z = \frac{-\left(Ax + By + D\right)}{C}, \text{ con } C \neq 0$$

Si en el píxel (x_i, y_s) se obtiene el valor z_i, y en el (x_{i+1}, y_s) el valor de z_{i+1}, el incremento en Z vendrá dado por:

ec. 17

$$\Delta z = z_{i+1} - z_i = \frac{-\left(Ax_{i+1} + By_s + D\right)}{C} + \frac{\left(Ax_i + By_s + D\right)}{C} = \frac{A\left(x_i - x_{i+1}\right)}{C}$$

Si decimos que $\Delta x = x_{i+1} - x_i$, queda:

ec. 18

$$\Delta z = \frac{A}{C}(-\Delta x) = -\frac{A}{C}\Delta x$$

Como

ec. 19

$$z_{i+1} = z_i + \Delta z$$

se necesita una simple resta para calcular z en el punto (x_{i+1}, y_s) a partir del valor de z en (x_i, y_s), ya que $-\frac{A}{C}\Delta x$ es constante.

Un cálculo incremental similar puede realizarse para determinar el valor de z al cambiar de línea de rastreo.

Método de interpolación de la profundidad

Alternativamente, si no se conoce la ecuación del plano, o no se desea utilizar el método previo, puede determinarse la profundidad de los píxeles de un segmento raster *interpolando las coordenadas z en los extremos del segmento.*

Este método es exactamente igual al de Gouraud, sólo que interpolando la profundidad, en lugar de las intensidades.

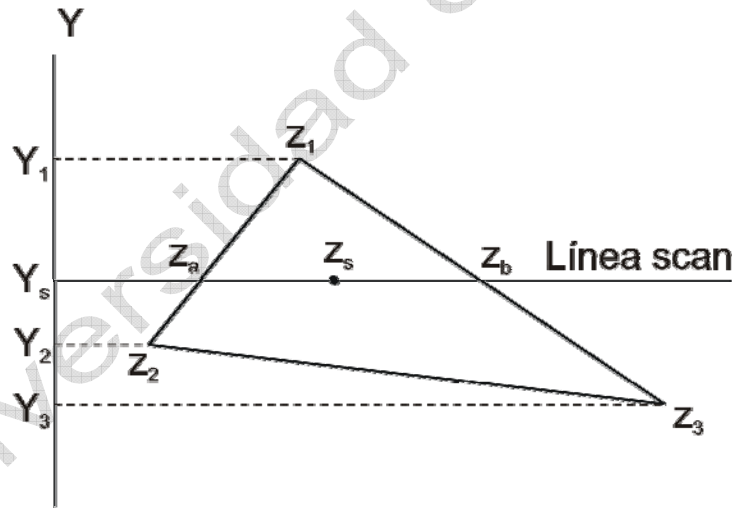


figura 18: interpolación de valores de z

Por lo tanto, si en las ecuaciones de Gouraud se sustituyen los parámetros de intensidad en los vértices y aristas ($I_1, I_2, I_3, \ldots, I_a, I_b$), por las correspondientes coordenadas Z $(z_1, z_2, z_3, \ldots, z_a, z_b)_{SRV}$ (figura 18), se obtienen las ecuaciones que proporcionan directamente la profundidad asociada a los puntos de los polígonos.

Sustituyendo en la ec. 12 (en Gouraud) queda:

$$z_{s,n} = z_{s,n-1} + \Delta z_s$$

Comparando esta ecuación con la ec. 19, vemos que ambos métodos son igualmente efectivos, pues basta una simple suma o resta para calcular la

profundidad del siguiente punto.

Características del algoritmo Z-buffer

- El Z-buffer es un algoritmo general y simple, que incluso puede ser utilizado con modelos no poliédricos.
- Realiza una ordenación en $(X, Y)_{SRPV}$ sin requerir comparación alguna, y para ordenar en Z realiza una comparación por píxel, por cada polígono que lo contiene.
- El tiempo necesario para determinar la visibilidad de las superficies *tiende a ser independiente del número de polígonos a visualizar*, ya que, en promedio, el número de píxeles cubiertos por cada polígono disminuye a medida que aumenta el número de polígonos en el volumen de visión.
- No impone limitaciones a la complejidad de las escenas. Una vez que se han establecido las dimensiones del espacio de la imagen, un aumento en la complejidad supone, como mucho, un incremento lineal del tiempo de cálculo.
- Otra gran ventaja es que la información registrada en el Z-buffer, al ser independiente de la imagen, puede utilizarse en muchos otros aspectos del rendering, como aislar objetos para visualizarlos, mezclar imágenes basándose en la profundidad de los objetos, etc.
- El tamaño de la memoria para registrar el Z-buffer, depende de la precisión con que se calcule la profundidad de cada punto (X, Y), que a su vez está en función de la complejidad de la escena (distancia mínima en Z entre los polígonos, y distancia de éstos al observador). Normalmente, *con 24 o 32 bits por registro son suficientes para obtener una precisión aceptable*. Con una resolución de 1024 x 1024, y usando 32 bits por píxel, se requiere un Z-buffer de 4 Mb. Esta "enorme" cantidad de memoria era, hasta no hace mucho, una de las principales desventajas del método. La disponibilidad actual de memoria ha eliminado o minimizado de golpe una serie de problemas, aún descritos en los principales libros de texto, cuyo origen estaba en la falta de precisión. Hoy día, hasta los ordenadores más cutres pueden permitirse el lujo de utilizar un Z-buffer de 8 Mb (64 bits por píxel), en caso de emergencia.
- Uno de los inconvenientes del algoritmo, aún vigente, es la dificultad que ofrece en la implementación de las técnicas anti-aliasing, transparencias, y efectos. La información necesaria para realizar el prefiltrado y anti-aliasing no es fácil de localizar debido a que el algoritmo escribe los píxeles en el buffer en orden aleatorio. Aunque estas técnicas son posibles, son difíciles de aplicar. Sin embargo, el postfiltrado es relativamente sencillo.

2.2 Discretización múltiple por línea scan

Según vimos, la discretización múltiple de los polígonos siguiendo la línea scan consiste en discretizar todos los polígonos de la escena que sean cruzados por una misma línea de rastreo, antes de cambiar a la siguiente.

Una de las mayores diferencias entre la discretización múltiple y la realizada polígono a polígono está en la forma de construir la lista de aristas. En la discretización múltiple aparecen dos problemas:

1) Si rasterizamos a priori todas las aristas de todos los polígonos que forman el poliedro se consumiría mucha memoria, limitando la complejidad de la escena. Para evitarlo, se mantiene una *lista de aristas activas*. Cuando comienza una nueva línea de rastreo, todas las aristas que empiezan en esa línea son añadidas a la lista, mientras que las que acaban en ella se borran. Para cada arista activa se almacenan la coordenada en X_{SRPV}, e información sobre la intensidad y sus incrementos. Cada vez que se añade una nueva arista, esos valores son inicializados. Mientas la arista se encuentre activa, los incrementos serán sumados con cada nueva línea de rastreo.

2) El otro problema que se plantea es el de saber a qué polígono pertenece cada arista activa, ya que normalmente existirán varios polígonos activos en una misma línea de rastreo. Por lo común, se ha de registrar información extra en la lista de aristas activas que las relacione con el polígono al que pertenecen. Su implantación depende mucho del algoritmo de eliminación de superficies ocultas utilizado. También es frecuente utilizar una *lista de polígonos activos*, donde se indican los polígonos que intersecan con la línea de rastreo en uso, y que por tanto, son los que pueden generar aristas activas. La lista de polígonos activos se actualiza con cada nueva línea scan, añadiendo nuevos polígonos y eliminando otros.

Si comparamos la estructura general del proceso de renderizado polígono a polígono, con la del renderizado múltiple tendremos una idea mejor de ambos. Cuando la discretización se hace polígono a polígono, la estructura del proceso es:

```
Para cada polígono
{
   • Construir la lista de aristas, a partir de las aristas del
     polígono;
   Para cada línea de rastreo
   {
      Para cada par de nodos (x_i, x_i+1) en la lista de aristas [y]
         • Discretizar el segmento raster de (x_i, y) a (x_i+1, y);
   }
}
```

En la discretización múltiple por línea scan, la estructura del proceso quedaría como sigue:

```
Inicializar la lista de aristas activas;
Para cada línea de rastreo
{
   Para cada arista que comienza en la línea scan actual
   {
      • Añadir arista a la lista de aristas activas;
      • Inicializar los valores de rasterización, intensidad e
        incrementos;
   }
   • Eliminar las aristas que finalizan en la línea scan;
   • Buscar en la lista de aristas activas los segmentos raster
     y mostrarlos;
   • Sumar los incrementos a todas las aristas activas;
}
```

Una de las ventajas del discretizado múltiple sobre el de polígonos individuales es que permite la utilización de un *Z-buffer de línea scan*. Este algoritmo de eliminación de superficies ocultas es una variación del Z-buffer que hemos visto para la discretización polígono a polígono. La principal diferencia entre ambos está en el tamaño del Z-buffer. Como su nombre indica, en el Z-buffer de línea scan *tiene tantos registros como píxeles haya en la línea de rastreo*, es decir, supone un gran ahorro de memoria.

Este algoritmo está en desuso, ya que no se valora el ahorro que representa en memoria y, por contra, obliga a que el Z-buffer deba ser inicializado por cada nueva línea scan, lo que conlleva pérdida de tiempo. Además, se ha de renunciar a las posibilidades gráficas que ofrece el disponer de la profundidad de cada píxel, una vez finalizado el rendering.

3. Algoritmo general del rendering standard

Hasta ahora hemos estado viendo los diferentes procesos del rendering de modo individual, sin ver claramente cómo encajan unos con otros. Para solventar esta deficiencia, veamos ahora el pseudocódigo de lo que podría ser el proceso general de rendering standard (versión pedagógica).

Condiciones iniciales

Inicialmente supondremos que:

- Los objetos, el visor y las fuentes se encuentran ubicados en el SUR.
- Por cada poliedro en el escenario se dispone de:
  - Matriz de conectividad de los vértices
  - Normal a la superficie de cada polígono
  - Normal media en los vértices
- Los objetos han sido modelados utilizando exclusivamente polígonos convexos.

Características del algoritmo de rendering

- Discretiza polígono a polígono
- Se usa el DDA para rasterizar las aristas
- Utiliza un Z-buffer para eliminar las superficies ocultas
- Aplica el método de intensidad de Phong
- Opcionalmente utiliza el método incremental de Gouraud o el de Phong, para dar intensidad a los polígonos.

```
Rendering_standard()
{
   • Inicializar el Z-buffer con el valor de profundidad máximo.

   //Se determina la visibilidad del polígono
   Para cada polígono en el escenario
   {
      • Aplicar el proceso de culling para eliminar las caras
        traseras no visibles;

      Si(polígono rechazado)
         • Pasar al siguiente polígono;
      sino
      {
         //Se obtiene la imagen vectorial y la profundidad
         Para cada vértice SUR(Xi, Yi, Zi) del polígono
         {
            SUR(Xi, Yi, Zi) -> SRPV(Xi, Yi);
            SUR(Xi, Yi, Zi) -> SRV(Zi);
         }

         Si(Gouraud)
            //Se calcula la intensidad en los vértices
            Para cada vértice SUR(Xi, Yi, Zi) del polígono
               • Calcular Ii aplicando el modelo de intensidad de Phong, utilizando la nor-
                 mal media (Ni) en el vértice;

         //Comienza el proceso de construcción de la matriz de aristas
         Para cada arista del polígono
         {
            Si(Gouraud)
               • Calcular el incremento de I (Δi), a partir de las intensidades Ii e Ij en
                 los vértices;
            sino
               • Calcular el incremento de la normal ΔN(ΔNx, ΔNy, ΔNz), a partir de las nor-
                 males medias Ni y Nj en los vértices;
            • Calcular el incremento de la profundidad (ΔZi), a partir de la profundidad
              SRV(Zi) y SRV(Zj) en los vértices;
            Mientras continúe la arista (Ymin != Ymax)
            {
               • Insertar el nodo correspondiente al punto SRPV(Xs, Ys) en la matriz de no-
                 dos (ordenando según XS), guardando en él, el valor de Is (o Ns) y
                 SRV(Zi).
               • Aplicando el DDA, se calcula el punto SRPV(Xs+1, Ys+1), correspondiente al
                 siguiente píxel de la arista.
               • Calcular los nuevos valores de I (o N) y SRV(Z), sumando los incrementos ΔI
                 (o ΔN) y ΔZ
            }
         }

         //Ya se han discretizado las aristas, y creado la lista de aristas
         //Ahora se procede al rellenado del polígono, siguiendo las líneas scan

         Para cada fila en la matriz de aristas
         {
            //Al ser todos los polígonos convexos, sólo habrá una pareja de nodos por fila
            • Calcular los incrementos ΔI (o ΔN) y ΔZ, a partir de (I1, I2)(o (N1, N2)) y
              (Z1, Z2), registrados en los nodos n1 y n2;
            • Inicializar I (o N) y Z con los valores de n1 o n2, según sea el sentido de
              trazado
            Mientras( (X1, Y) != (X2, Y) )
            {
               Si(Z <= Z_buffer[X, Y])
               {
                  //El punto está, de momento, más cercano al observador que ningún otro
                  Z_buffer[X, Y] = Z;
                  Si(Gouraud)
                     buffer_pantalla[X, Y] = I;
                  sino
                     buffer_pantalla[X, Y] = Calcular_intensidad(N);
               }
               • Incrementar X, I (o N) y Z;
            }
         }
      }
   }
} /* fin Rendering_standard() */
```

algoritmo 6: rendering standard general