Organización del Computador II

Librería Gráfica

Desarrollo Teórico

1er cuatrimestre de 2001

Índice

1. Herramientas de Dibujo

1.1. Método Bresenham para rectas

Este método consiste principalmente en evaluar el error que representa la distancia entre la recta real y un punto graficado.
La idea básica del algoritmo es reemplazar la pendiente real de la recta valuada por sus componentes enteros. Es decir, no es necesario calcular la pendiente de la recta (este es el caso de otros algoritmos de graficación de rectas, como el DDA), para decidir que píxeles prender.
En palabras mas simples, si uno quiere graficar la línea de la **figura 1.1**, sean los píxeles **P**, **D** y **E**, se quiere saber cual es la mejor aproximación a la línea **y**. Se comienza con un error inicial de 0 ($e_o = 0$), pero luego a medida que se va avanzando en el algoritmo, el error va cambiando. Supongamos que en el píxel **P** hay un error de e, se elegimos el píxel E, el nuevo error es:

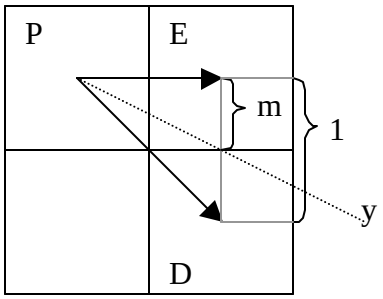


Figura 1.1. *representación de una línea Bresenham*

$$e = e + m$$

mientras que si elegimos el píxel D, el nuevo error es:

$$e = e + (1 - m)$$

entonces, si e + m > 0.5 (la distancia de medio píxel), elegimos dibujar D.
El objetivo de este cálculo es la rapidez. Queremos que todas las cuentas sean con enteros, para que las operaciones no sean costosas.

La idea implementada es
Sean $x_0, y_0, x_1, y_1 \in N$; $\Delta x = x_1 - x_0$, $\Delta y = y_1 - y_0$.

$e_o = m - ½ = 2 * (\Delta y / \Delta x) - 1$

si e>0 , entonces elijo el píxel **D**.

Pero para lograr que todas las cuentas sean enteras, multiplicamos todo por $2*\Delta x$.
Entonces:
Si elegimos E, $e = e + 2\Delta x$
Si elegimos D, $e = e + 2\Delta y - 2\Delta x$

Este algoritmo se divide en cuatro casos, dependiendo del valor de la pendiente de la recta (en realidad son ocho, pero lo reducimos a cuatro casos por que podemos intercambiar el píxel de salida con el de llegada).

El algoritmo del caso base es, con las entradas (*x0, y0*) el píxel inicial, (*x1,y1*) el píxel final, *color* el color de la línea.

```
Línea (x0, y0, x1, y1, color)
        Δx ← x1-x0
        Δy ← y1-y0
        ix ← 2*Δx
        iy ← 2*Δy
        y ← y0                                  A1
        e ← iy - Δy
        para x = x0 hasta x1 hacer              A2
                PutPixel(x, y, color)
                e ← e + iy                      A3
                si e > 0 entonces
                        y ← y + 1               A4
                        e ← e – ix              A5
                fin si
        fin para
fin
```

Casos: (donde m es la pendiente, tal que $m=\Delta y/\Delta x$)

1. $0 < m < 1, \Delta x \geq 0, \Delta y \geq 0$

Este es el caso base. No hay cambios.

2. $m > 1, \Delta x \geq 0, \Delta y \geq 0$

En este caso el algoritmo sufre los siguientes cambios.
A1: x ← x0
A2: **para** y = y0 **hasta** y1 **hacer**
A3: e ← e + ix
A4: x ← x + 1
A5: e ← e – iy

3. $-1 < m < 0, \Delta x \geq 0, \Delta y < 0$

En este caso los cambios en el algoritmo base son.
A3: e ← e – iy
A4: y ← y – 1

4. $m > -1, \Delta x \geq 0, \Delta y < 0$

y para esta variedad, los cambios son
A1: x ← x0
A2: **para** y = y1 **hasta** y0 **hacer**
A3: e ← e + ix
A4: x ← x – 1
A5: e ← e + iy

Otros casos (deben implementarse para un algoritmo de línea completo).

a. $\Delta x < 0$.

Cuando esto pasa se intercambian :

- (x0,y0) con (x1,y1)
- Δx con $-\Delta x$
- Δy con $-\Delta y$

y se utiliza uno de los casos normales de **1.** a **4.** .

b. $\Delta x = 0$ (m no está definido).
Quiere decir que la línea es vertical, no se necesita hacer un algoritmo complicado.

c. $\Delta y = 0$ (m = 0).
Quiere decir que la línea es horizontal, no se necesita hacer un algoritmo complicado.

1.2. Discretización de Polígonos

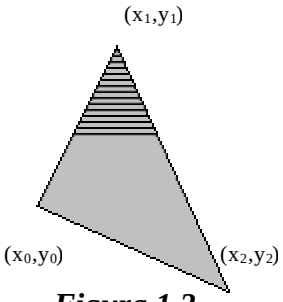


Figura 1.2. *polígono a dibujar*

Usamos un método relativamente simple para rellenar triángulos, ya que todo polígono es representable a partir de finitos triángulos.
Para hacer esto hay que definir 2 arreglos auxiliares:

minx[0..AltoDePantalla]
maxx[0..AltoDePantalla]

y obtener también el mínimo de y (*miny*) y el máximo de y (*maxy*), para saber entre que puntos hay que iterar.

Con todos estos datos, lo que se hace es, por ejemplo en la figura 1.2., obtener en el arreglo *minx*, en las posiciones y_1 a y_0, los valores de *x* de la recta $((x_0, y_0), (x_1,y_1))$, en las posiciones y_0 a y_2, los valores de *x* de la recta $((x_0, y_0), (x_2,y_2))$, *y* en el arreglo *maxx*, en las posiciones y_1 a y_2, los valores de *x* de la recta $((x_1, y_1), (x_2,y_2))$. Con todos estos datos, luego se dibujan las rectas definidas por *(minx[y],y)* , *(maxx[y],y)*, para todo *y* entre *miny* y *maxy*, tal como muestra la figura 1.3.

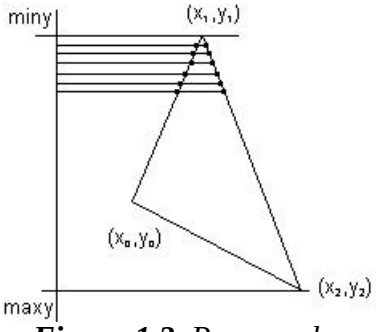


Figura 1.3. *Proceso de dibujo*

Para obtener los resultados buscados, se separan los siguientes casos, sobre los valores sobre las coordenadas *Y* de los puntos (x0, y0), (x1,y1), (x2,y2). Para esto se tiene una variación de la función línea (llameémosla LineaModificada), que recibe un arreglo *A* como parámetro, y en vez de graficar el punto (*X*, *Y*) en la pantalla, escribe en *A*[*Y*] el valor *X*.

1. Se definen, comparando y_0, y_1 e y_2, los valores *miny* y *maxy*, que me indican el mínimo y el máximo valor sobre y.
2. Se procesa los valores de *minx* y *maxx*, los arreglos antes nombrados. Para esto se ven los siguientes casos.
   - $y_0 = y_1 = y_2$: En este caso solo se debe dibujar una línea entre el mínimo y el máximo valor de *X*.
   - $y_0=y_1 \wedge y_0 \neq y_2$. Para este caso se llama al función se definen los arreglos de la siguiente manera:

LineaModificada(x_0, y_0, x_2, y_2, *maxx*)
LineaModificada(x_1, y_1, x_2, y_2, *minx*)

- $y_0 = y_2 \wedge y_0 \neq y_1$:
  LineaModificada(x_0, y_0, x_1, y_1, *maxx*)
  LineaModificada(x_2, y_2, x_1, y_1, *minx*)
- $y_1 = y_2 \wedge y_0 \neq y_1$:
  LineaModificada(x_1, y_1, x_0, y_0, *maxx*)
  LineaModificada(x_2, y_2, x_0, y_0, *minx*)
- $(y_1 = miny \wedge y_2 = maxy) \vee (y_2 = miny \wedge y_1 = maxy)$:
  LineaModificada(x_1, y_1, x_2, y_2, *maxx*)
  LineaModificada(x_1, y_1, x_0, y_0, *minx*)
  LineaModificada(x_0, y_0, x_2, y_2, *minx*)
- $(y_0 = miny \wedge y_2 = maxy) \vee (y_2 = miny \wedge y_0 = maxy)$:
  LineaModificada(x_0, y_0, x_2, y_2, *maxx*)
  LineaModificada(x_1, y_1, x_0, y_0, *minx*)
  LineaModificada(x_1, y_1, x_2, y_2, *minx*)
- $(y_0 = miny \wedge y_1 = maxy) \vee (y_1 = miny \wedge y_0 = maxy)$
  LineaModificada(x_0, y_0, x_1, y_1, *maxx*)
  LineaModificada(x_1, y_1, x_2, y_2, *minx*)
  LineaModificada(x_2, y_2, x_0, y_0, *minx*)

3. Para todo *Y* entre *miny* y *maxy* se hace:
   Linea(*minx*[Y], Y, *maxx*[Y], Y, *colorPoligono*)
   Donde *colorPoligono* es el parámetro que indica el color

2. Herramientas de Sombreado

2.1. Paletas de Colores Uniformes

El sistema de colores RGB, basado en el sistema visual del ojo humano, representa todos los colores con ciertas proporciones del Rojo, Verde y Azul (todos estos saturados). Al comparar las intensidades en una fuente de luz, percibimos el color de la luz.
Se dice que el RGB es un modelo aditivo ya que se suman las intensidades de los colores primarios para producir otros colores.
De esta manera un color se puede representar de la siguiente manera:

$$C_\lambda = rR + gG + bB$$

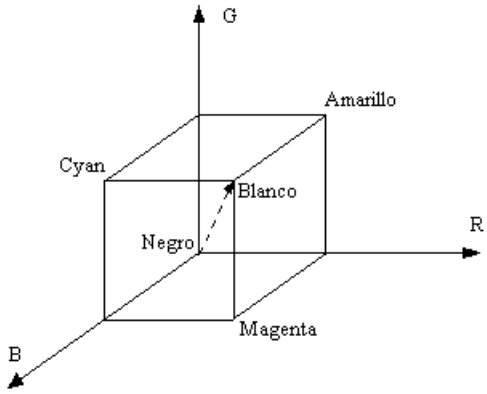


Figura 2.1. *Representación de RGB*

Con este esquema es posible obtener un número mínimo de colores. Con 3 bits por pixel se utiliza cada una de las tres posiciones para controlar el nivel de intensidad de cada uno de los tres colores.

Para obtener una paleta de colores que represente mejor el espectro de colores con solo 256 colores, se busca establecer una paleta que abarque todos los colores en una manera ordenada. Esto es necesario debido a que si uno quiere sombrear, es necesario tener una progresión coherente sobre la cual iterar (al no tener una escala completa de RGB).
Para obtener este tipo de paleta se hace una partición sobre la cantidad de colores. Se usan generalmente dos paletas diferentes para este propósito. Una con 256 colores y otra con 252 colores.
La paleta de 256 colores utiliza 8 tonalidades de Rojo, 8 tonalidades de Verde y 4 tonalidades de azul, conformando un total de 8*8*4 = 256 colores. La paleta de 252 colores es mas estable en la cantidad de tonalidades, utiliza 6 tonalidades de Rojo, 7 tonalidades de Verde y 6 tonalidades de Azul. Nótese que siempre el color verde aparece mas, esto se debe a que en este color recae la mayoría de la luminosidad (aproximadamente 60% del total).
Estas paletas, se utiliza lo siguiente (El ejemplo se realiza sobre la segunda, la primera es muy parecida de construir).
Se crean tres arreglos:
Rojo [0,50,100,150,200,255]
Verde [0,50,100,140,180,220,255]
Azul [0,50,100,150,200,255]

Y luego se crea la paleta con el siguiente algoritmo.

```
para i = 0 hasta 5 hacer
    para j = 0 hasta 6 hacer
        para k = 0 hasta 5 hacer
            setRGBpalette(i+6*j+42*k, Rojo[i], Verde[j], Azul[k])
```

donde los parámetros de setRGBpalette son:
setRGBpalette(índice en la paleta, intensidad rojo, intensidad verde, intensidad Azul).

Con esta algoritmo la paleta toma el siguiente aspecto:

Tabla 2.1 – *Resultado de la paleta estable*

Índice	Intensidad Rojo	Intensidad Verde	Intensidad Azul
0	0	0	0
1	50	0	0
...	...	...	....
5	255	0	0
6	0	50	0
7	50	50	0
8	100	50	0
...	...	...	...
41	255	255	0
42	0	0	50
43	50	0	50
44	100	0	50
...	...	...	...
251	255	255	255
252	255	255	255
253	255	255	255
254	255	255	255
255	255	255	255

Estos colores se repiten para no tener colores no definidos.

2.2. Sombreado de Gouraud

Con el método de sombreado suave de Gouraud no es necesario tomar una muestra de la iluminación local de cada píxel ya que se realiza una interpolación de la intensidad (o sea, no se compliquen con modelos de iluminación). Principalmente consiste en calcular el modelo de iluminación de cada vértice (o en nuestro caso, decidir que color hay en cada vértice), y a partir de ellos interpolar el valor de iluminación a lo largo de las aristas durante la conversión scan del polígono.

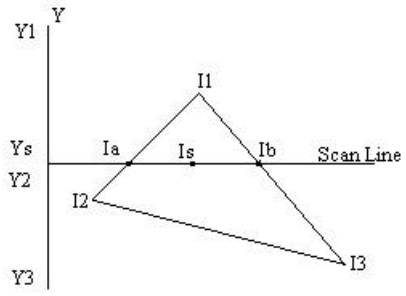


Figura 2.2. *método de sombreado*

Siguiendo el dibujo anterior, el algoritmo del método calcula la iluminación de los vértices *I1*, *I2*, *I3*. Luego, interpolando entre *I1* e *I2* se obtiene Ia, y entre *I1* e *I3* se obtiene Ib. Para cada línea de barrido se interpola nuevamente entre Ia e Ib, obteniendo Is.

Figura 2.3. *resultados del método*

Dato: En un contexto de modelos de iluminación, la desventaja que presenta este método es que realiza aproximaciones poco precisas cuando el polígono a sombrear es iluminado por una fuente de luz muy cercana, ubicada en su centro. Esto se debe a que al tomar la iluminación de los vértices, éstos son

similares y todo el polígono será pintado de un color constante. Como solución a este problema se puede subdividir el polígono en otros más pequeños.
Otra desventaja que presenta es que se produce una distorsión en el efecto de la perspectiva (también sucede si el objeto es rotado). Esto significa que la interpolación no es invariante frente a transformaciones sobre un modelo de iluminación.

Para este algoritmo se necesitan dos arreglos tal como se necesita en la discretización de polígonos. En este caso se necesita guardar, aparte de su posición en *X* y los componentes *R*, *G* y *B* de su color.
Tal como el método descripto en el punto **1.2. Discretización de Polígonos**, se dibuja un polígono con una variante sobre los algoritmos de línea.
Las líneas se dibujan con un incremento sobre las componentes Roja, Azul y Verde de los colores de los puntos que definen la línea. Siguiendo el algoritmo de Bresenham para líneas descripto en el punto **1.1. Método Bresenham para rectas**, se agregan (para todas las variantes detalladas), el siguiente código (las funciones Rojo(x,y), Verde(x,y) y Azul(x,y) me devuelven las componente roja, azul o verde del punto (x,y)). Se usan unas variables r, g y b, de tipo entero, que se inicializan con las componentes Roja, verde y azul del punto inicial. Los casos que hay que separar son:

- $\Delta x > \Delta y$
  - **A1:** $\Delta r \leftarrow$ (Rojo(x1,y1)-Rojo(x0,y0))/ Δx
    $\Delta g \leftarrow$ (Verde(x1,y1)-Verde(x0,y0))/ Δx
    $\Delta b \leftarrow$ (Azul(x1,y1)-Azul(x0,y0))/ Δx
  - Al Dibujar el Píxel va:
    PutPixel (x, y, color(r, g, b))
    donde color es una función que aproxima al color mas adecuado sobre la paleta
  - Al terminar el condicional (**si** ... **entonces** ... **sino** ... **finSi**) se agrega:
    $r \leftarrow r + \Delta r$
    $g \leftarrow g + \Delta g$
    $b \leftarrow b + \Delta b$
- $\Delta y \geq \Delta x$
  - **A1:** $\Delta r \leftarrow$ (Rojo(x1,y1)-Rojo(x0,y0))/ Δy
    $\Delta g \leftarrow$ (Verde(x1,y1)-Verde(x0,y0))/ Δy
    $\Delta b \leftarrow$ (Azul(x1,y1)-Azul(x0,y0))/ Δy
  - Al Dibujar el Píxel va:
    PutPixel (x, y, color(r, g, b))
    donde color es una función que aproxima al color mas adecuado sobre la paleta
  - Al terminar el condicional (**si** ... **entonces** ... **sino** ... **finSi**) se agrega:
    $r \leftarrow r + \Delta r$
    $g \leftarrow g + \Delta g$
    $b \leftarrow b + \Delta b$

La función *Color* hace uso de la paleta de colores estables para aproximar de manera rápida el color que esta sobre la paleta con el deseado.

Entrada: r, g, b enteros

```
Variables auxiliares: i, j, k  enteros
Color
        r ← r/256
        g ← g/256
        b ← a/256
        r ← r*63
        i ← 0
        Si r > 0 entonces
                mientras (rojo[i]<r) hacer i ← i+1 finMientras
        Si random(rojo[i]-rojo[i-1]) > (r - rojo[i-1] ) entonces i ← i-1 finSi
        finSi
        g ← g*63
        j ← 0
        Si g > 0 entonces
                mientras (verde[j]<g) hacer j ← j+1 finMientras
        Si random (verde[j]-verde[j-1]) > (g-verde[j-1] ) entonces j ← j-1 finSi
        finSi
        b ← b*63
        k ← 0
        Si b > 0 entonces
                mientras (azul[k]<b) hacer k ← k+1 finMientras
        Si random(azul[k]-azul[k-1]) > (b-azul[k-1] ) entonces k ← k-1 finSi
        finSi
        devolver(i+6*j+42*k)
fin
```

Las partes marcadas corresponden a un método llamado *dither*. Esta implementación será explicada en el siguiente punto.

2.3. Interpolación Aleatoria (Dithering)

Este método es usado para dar mayor realismo. Es similar al efecto usado en pinturas al óleo pertenecientes a la escuela de puntillismo.

Se busca generar que al mostrar un objeto con una transición de colores, estos estén mezclados en cierta medida. Esto se logra mezclando los colores con aquellos de sus alrededores.

Tal como se ha probado en el punto anterior, es una técnica fácil de implementar, donde se obtienen resultados mas realistas con menor costo. Por ejemplo, en las figuras **2.3** y **2.4**, para obtener un resultado parecido al dithering sin aplicarlo, se debería partir el triangulo en subtriangulos mas pequeños y utilizar el método de sombreado sobre todos ellos. Cuantas mas iteraciones se hagan sobre los triángulos, mas es la calidad del realismo. Esto es muy costoso en comparación de una función de *random*.

Figura 2.4. *Sin Dithering*

Figura 2.5. *con Dithering*

3. Bibliografía.

Las referencias [1] y [3] se encuentran en la Infoteca. La referencia [2] se encuentra en la fotocopiadora del pabellón 1, en la carpeta de Computación Gráfica.

[1] Computación Gráfica por Computador.
Marc Berger. Adisson-Wesley. 1991.

[2] Computación Gráfica.
Claudio Delrieux

[3] Gráficas por Computadora.
Donald Hearn - M. Pauline Baker. Prentice-Hall. 1995 Segunda edición.

I. Apéndice A, Implementaciones en MMX.

I.1. Algoritmo de Sombreado de Gouraud.

Este segmento fue extraído del sitio http://developer.intel.com/.

I.1.1. Introduction

The Intel Architecture (IA) media extensions include single-instruction, multi-data (SIMD) instructions. This application note presents examples of code that exploit these instructions. Specifically, the Gouraud Shading algorithm presented here illustrates how to use the new MMX^{TM} technology instructions to achieve better performance in color rendering. The performance improvement relative to traditional IA code can be attributed primarily to the technique of processing multiple data elements in parallel.

I.1.2. The Gouraud shading algorithm

Gouraud shading is a scan line algorithm used to render objects smoothly in three-dimensional (3D) graphics. If a scan line algorithm is used to render an object, a value for the intensity of each pixel along the scan line must be determined from the illumination model. In implementation, the object is divided into polygonal surfaces. Each individual polygonal surface is rendered as a sequence of scan lines. Gouraud shading first determines the intensity at each polygonal vertex, then uses a bilinear interpolation to determine the intensity of each pixel on a scan line.

Figure A.1. Bilinear Interpolation

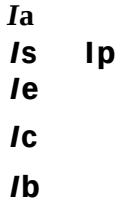


Figure 1 and the formulas shown in Figure 2 illustrate how this is done. It is assumed that ***S*** and ***E*** are intersection points of a scan line and a triangle (Figure 1).

Figure A.2. Bilinear Interpolation Formulas

$$\boldsymbol{I}s = v * \boldsymbol{I}a + (1 - v) * \boldsymbol{I}b \quad 0 <= v <= 1 \quad (1)$$

$$\boldsymbol{I}e = t * \boldsymbol{I}a + (1 - t) * \boldsymbol{I}c \quad 0 <= t <= 1 \quad (2)$$

$$\boldsymbol{I}p = u * \boldsymbol{I}s + (1 - u) * \boldsymbol{I}e \quad 0 <= u <= 1 \quad (3)$$

$$\boldsymbol{I}p(i+1) = \boldsymbol{I}p(i) + d\boldsymbol{I} \quad i = 1,2,\ldots n-1 \quad (4)$$

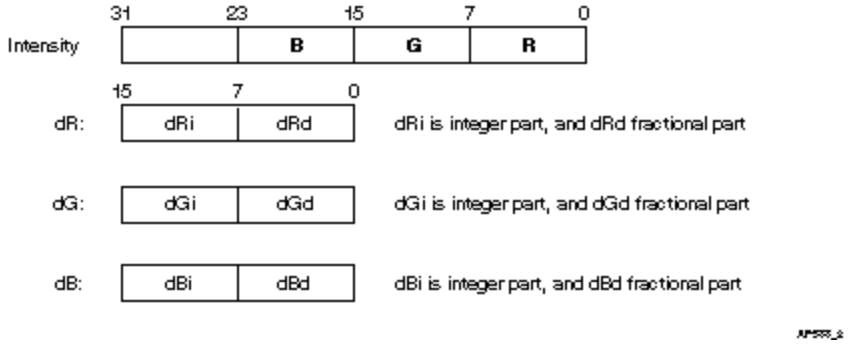
$$d\boldsymbol{I} = (\boldsymbol{I}e - \boldsymbol{I}s)/(n-1) \quad (5)$$

The bilinear interpolation is performed in two steps:
1. Intensities ***Is***, ***Ie*** at the pixels ***S*** and ***E*** are determined by linearly interpolating the intensities of the triangle vertices. This is shown in Formulas 1 and 2.
2. The intensity ***I***p at a pixel ***P*** on the scan line is also obtained by linearly interpolating along the scan line between ***S*** and ***E***, as shown in Formula 3.
The example code in this application note performs the second step of linear interpolation as shown in formula 3. Because addition takes fewer cycles than multiplication, if addition is the only operation necessary, Formula 4 is used instead of Formula 3.
Where ***I***p(i) is the intensity of the *i*th pixel, n is the number of the pixel on the scan line, d***I*** is the incremental intensity for each successive pixel. Formula 5 shows how this is calculated.

I.1.3. Input and Output data representation

The intensity of a pixel is represented by a 32-bit DWORD in which RGB values, each represented by eight bits, are stored in the first 24 bits. In order to reduce loss of accuracy due to truncation resulting from Formula 5, the incremental values of *dR*, *dG*, and *dB* are represented as a fixed-point notation stored in a 16-bit SWORD. The fractional part is stored in the first eight bits and the integer part in the remaining eight bits. Refer to Figure 3 for further clarification.

Figure A.3. Format of Input/Output Data



We assume that the ranges of RGB values are from 0 to 255 and that there are at least four pixels on the scan line. Hence the ranges of *dR*, *dG* and *dB* values are from -255/4 to 255/4. The resulting intensities are stored in an array of DWORD. If there are fewer than four pixels on the scan line, there is little performance benefit from coding in assembly language. C language code should be used instead.

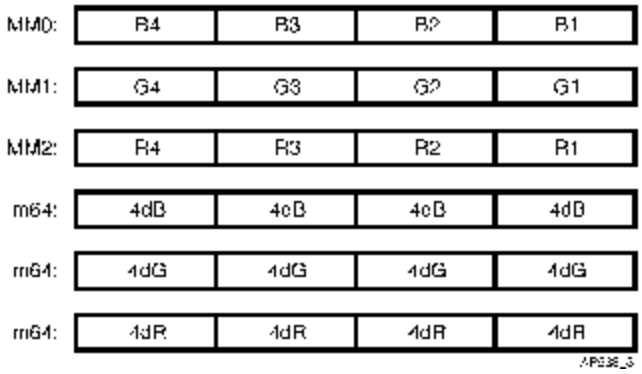
I.1.4. Code partitioning

The procedure primarily consists of two main sections: a data setup section and an inner loop section in which intensities are assigned to four pixels. In both sections, the code fully exploits SIMD techniques to process four intensities in parallel.

I.1.4.1. Flow in Data Setup

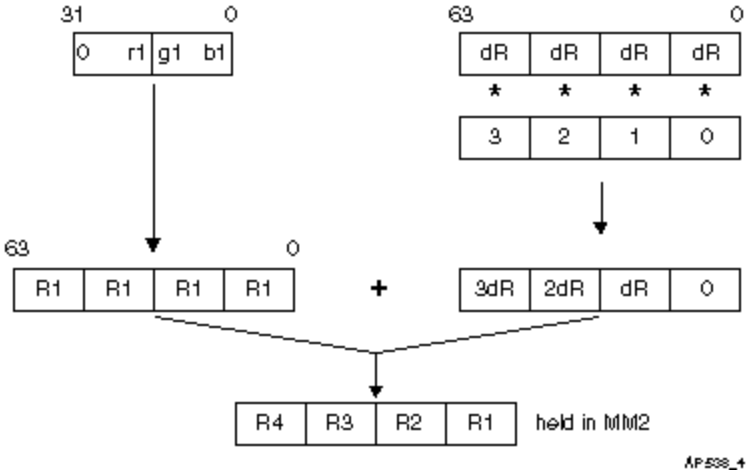
In the data setup section, the task is to prepare data as shown in Figure 4 for the inner loop. This data will be stored in MMX^{TM} registers or memory locations.

Figure A.4. Data Setup



The MMX registers MM2, MM1 and MM0 hold RGB values for four consecutive pixels, where *R2 = R1 + dR,... R4 = R3 + dR* ... etc. *R1, R2, R3, R4*, each 16 bits, serve as accumulators. Like *dR*, color values of *R1, R2, R3, R4*, are also represented in fixed-point notation. Four times the incremental color values *dR*, *dG* and *dB* are stored in four consecutive memory locations and are used for accumulation of color values for R,G and B in the inner loop. This setup code allows us to evaluate Formula 4 for four adjacent pixels in parallel. The computation flow for *R1, R2, R3, R4* (shown in Figure 4) is shown in Figure 5.

Figure A.5. Flow of Computing R1, R2, R3, R4 and Loading Into MM2



An alternative approach would have been to prepare registers that contain [0,B1,G1,R1] and [0,dB,dG,dR]. This approach was rejected because the inner loop would only do three calculations in parallel instead of four. Hence this alternative has poor performance for scan lines with many pixels, where performance is needed most.
Example 1 shows a sequence of MMX instructions that extract the R color value (r1 in Figure 5) from RGB1, and load r1 to the higher byte of four words in MM2. Here RGB1 is a 32-bit input data that holds the intensity of the first pixel on the scan line.

Example A.1. Extracting R From Rgb1 and Loading R to the First Byte of MM2

```
movdt mm6, Rgb1          ; load R1,G1,B1 intensities to lower three bytes of mm6
punpcklbw mm6,mm6        ; unpack R1R1,G1G1,B1B1 to the first 3 words of mm6
movq mm7, mm6            ; load R1R1,G1G1,B1B1 to the first 3 words of mm7
punpcklwd mm7,mm7        ; unpack R1R1,R1R1,G1G1,G1G1 to the four words of mm7
pxor mm2,mm2             ; clear mm2 for use by unpack
punpcklbw mm2,mm7        ; unpack R1 to the higher byte of the four words of mm2
```

A similar set of instructions can be used to form [G1,G1,G1,G1] and [B1,B1,B1,B1]. Each of these three sets of instructions do not pair well because many of the instructions operate on the result of the immediately preceding instruction. However, by interleaving these three groups of instructions, the resulting code pairs very well. (See I.1.6. Implementation)

I.1.4.2. Flow in Inner Loop Section

There are two tasks to be accomplished in the inner loop: calculating intensities for four consecutive pixels and updating MM0, MM1, and MM2. Computation flow for the parallel processing of data in MM0, MM1, and MM2 into four 32-bit RGB intensities is shown in Figure 6. MM2, MM1, and MM0 serve as accumulators of RGB color values. Their initial values are calculated in the data setup section described in section 4.1. Their values are updated by adding the incremental dR, dG and dB for each pass through the inner loop as shown in Figure 7.
As mentioned in section 4.1, R1, R2, R3, and R4 are represented in fixed-point notation with the first 8 bits representing the fractional part and the remaining 8 bits the integer part.

Values r1, r2, r3, and r4 shown in Figure 6 are the integer parts extracted from R1, R2, R3, and R4 and similarly for g1, g2, g3, and g4 and b1, b2, b3, and b4.
Since we process four pixels at a time, we need to handle the remainder if the number of pixels on a scan line is not a multiple of four. One approach is to process the remainder outside the inner loop. The drawback of this approach is that it needs four extra conditional jump instructions to differentiate four possible values of remainder, which would be 0,1,2, or 3. If done this way, the total cycles needed to process the remainder outside the inner loop would be more than 10.
A better approach is to handle the remainder in the inner loop. In this implementation, the output is stored into an array, and the calling routine transfers the pixels to the display. Since the caller can simply ignore unwanted pixels, the only cost is the extra allocated space to hold the extra pixels.

Figure A.6. Packing of RGB Into Four 32-Bit Pixel Intensities

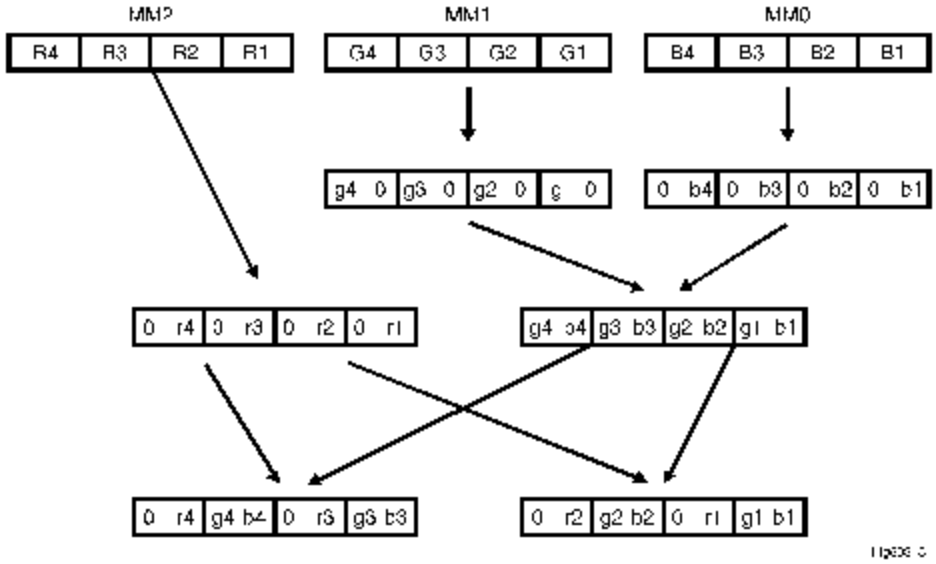
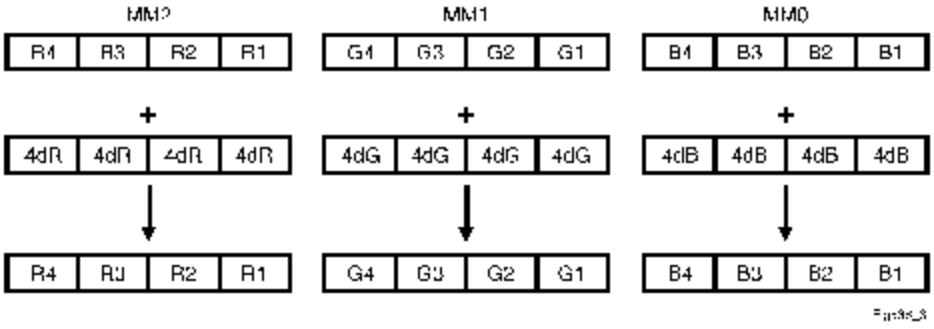


Figure A.7. Updating Accumulators in MM0, MM1, and MM2



The following MMX technology instructions implement the flow described in Figure 6, that is, packing RGB color values from MM0, MM1, and MM2 into four 32-bit intensities: R1G1B1, R2G2B2, R3G3B3 and R4G4B4. MM7 in Example 4 has been assigned 0ff00H

in its four words during the data setup section. It serves as a mask to clear the fractional parts of G1,G2,G3,G4 in MM4.

Example A.2. MMX™ Instructions to Implement Figure 5

```
movq      mm4, mm1        ; copy G1,G2,G3,G4 into four words of mm4
pand      mm4, mm7        ; clear the fractional part of G1,G2,G3,G4 in mm4
movq      mm3, mm2        ; copy R1,R2,R3,R4 into >four words of mm3
psrlw     mm3, 8          ; shift the integer parts of
                          ; R1,R2,R3,R4 to lower byte.
por       mm3, mm4        ; combine G1R1,G2R2,G3R3,G4R4 to four words of mm3
movq      mm6, mm3        ; copy G1R1,G2R2,G3R3,G4R4 to mm6.
movq      mm5, mm0        ; copy B1,B2,B3,B4 into four words of mm5
psrlw     mm5, 8          ; shift the integer parts of
                          ; R1,R2,R3,R4 to lower byte.
punpcklwd mm3, mm5        ; unpack B1G1R1, B2G2R2 to 2 dwords of mm3
punpckhwd mm6, mm5        ; unpack B3G3R3, B4G4R4 to 2 dwords of mm6
```

I.1.5. MMX™ technology performance

After pairing to optimize the MMX code, it takes 32 clocks in data setup and 10 clocks for each pass through the inner loop. Each pass processes intensities for four pixels. Table 1 shows how the number of clocks needed ɒ process one pixel varies with the number of pixels on a scan line and shows how the number drops dramatically as the number of pixels increase.

Table A.1. Number of Clocks Needed to Process a Pixel

No. of Pixels	Total Clocks	Clock Per Pixel
4	50	12.8
8	60	7.5
12	70	5.8
16	80	5.0
20	90	4.5
40	140	3.4

I.1.6. Implementation

The following code example is MMX code that implements Formula 4 for Gouraud Shading. The code has been optimized by appropriate pairing.

```
;************************************************************
; Procedure Scan_ln_RGB_MMX
;
; Description:
; Procedure Scan_ln_RGB_MMX implements a RGB intensities
; interpolation on a scan line, which is a major part of the
; Gouraud shading algorithm. The function of Scan_ln_RGB_MMx
; is, knowing the RGB intensity of the starting pixel of a
; scan line and the incremental R,G,B values, calculating
; the RGB intensity for each pixel on the scan line by
; linear interpolation.
;INPUTS:
;   Rgb1(DWORD): RGB intensity of the starting pixel on the scan ;line. Color
;values of R1,G1,B1 (8 bits each) are stored in first 24 ;bits.
```

```
;   dR(SWORD):    Incremental intensity of R for each successive pixel. dR
;is a 16-bit fixed point notation with the first 8 bits representing the
;fractional part and the remaining 8 bits the integer part. In the
;procedure, accumulation of dR includes both integer part and fractional
;parts, but only the integer part is used as when assigning RGB intensity
;to a pixel.
;   dG(SWORD):    refer to dR
;   dB(SWORD):    refer to dR
;   NumP(WORD):   number of pixels on the scan line.
;
; OUTPUTS:
;  Rgbs(PTR DWORD):       RGB intensities of all pixels on the scan line.
;Intensities R,G,B(each 8 bits)are stored in first 24 bits in a DWORD. In
;the procedure, R,G,B are calculated by linear interpolation, eg.
;        R(i) = R1 + i*dR,
;        G(i) = G1 + i*dG,
;        B(i) = B1 + i*dB.
;        i = 0, ..., NumP - 1.
; Assumption:     Input 'NumP' is greater than 3. The case that NumP is
;fewer than 4, is handled in C code.
;*************************************************
.486P
.MODEL FLAT, C
include iammx.inc
.Data
.Code
Public Scan_ln_RGB_MMX
Scan_ln_RGB_MMX Proc C Public USES eax ebx ecx edi,
Rgb1: DWORD,        ; RGB intensities of the start pixel.
Rgbs: PTR DWORD,    ; output, RGB intensities of all pixels.
dR:       SWORD,    ; increment of red intensity.
dG:       SWORD,    ; increment of green intensity.
dB1:      SWORD,    ; increment of blue intensity.
NumP:     WORD      ; number of pixels on the scan line.
;Declare local variables:
LOCAL dRed[4]:      SWORD     ; reserve 4 words for holding
                              ; increment of red.
LOCAL dGreen[4]:    SWORD     ; reserve 4 words for holding
                              ; increment of green.
LOCAL dBlue[4]:     SWORD     ; reserve 4 words for holding
                              ; increment of blue.
LOCAL factor:       QWORD     ; reserve 4 words for holding
                              ; 4 factor values
; load dR to 4 words of mm3,
; load dG to 4 words of mm4,
; load dB to 4 words of mm5
; Pack R1,G1,B1 from input Rgb1
; Load R1, R1+dR, R1+2*dR, R1+3*dR as 4 words in mm2
; Load G1, G1+dG, G1+2*dG, G1+3*dG as 4 words in mm1
; Load B1, B1+dB, B1+2*dB, B1+3*dB as 4 words in mm0
movdt       mm3, [ebp + 16]   ; load dR in the lowest word in  mm3
pxor        mm2,mm2           ; clear mm2
movdt       mm4, [ebp + 20]   ; load dG in the lowest word in mm4
punpcklwd   mm3,mm3           ; unpack dR to lower 2 words in mm3
movdt       mm5, [ebp + 24]   ; load dB in the lowest word in mm5
punpcklwd   mm4,mm4           ; unpack dG to lower 2 words in mm4
lea         ecx, factor       ; Load offset address of dRed.
Punpckldq   mm3,mm3           ; unpack dR to 4 words in mm3
mov         eax, 010000h
punpcklwd   mm5,mm5           ; unpack dB to lower 2 words in mm5
movdt       mm6, Rgb1         ; load R1,G1,B1 in lower three bytes to mm6
punpckldq   mm4,mm4           ; unpack dG to 4 words in mm4
mov         [ecx], eax        ; store values 0,1(each 2 bytes) in
```

```
                                ; memory location 'ecx'
punpcklbw  mm6,mm6              ; unpack R1R1,G1G1,B1B1 to the first 3
                                ; words of mm6.
mov        eax, 030002h
punpckldq  mm5,mm5              ; unpack dB to 4 words in mm5
movq       mm7, mm6             ; load R1R1,G1G1,B1B1 to the first
                                ; three words of mm7.
pxor       mm1,mm1              ; clear mm2
mov        [ecx + 4], eax       ; store values 2,3(each 2 bytes) in
                                ; memory location 'ecx + 4'
punpcklwd  mm7,mm7              ; unpack R1R1,R1R1,G1G1,G1G1 to 4
                                ; words of mm7.
Movq       mm0, mm3             ; unpack R1 to the higher bytes of the
                                ; 4 words of mm7
punpcklbw  mm2,mm7              ; unpack R1 to the higher bytes of
                                ; the 4 words of mm2
pmullw     mm0, [ecx]           ; multiplied by [3,2,1,0], mm0
                                ; becomes [3dR,2dR,dR,0]
punpckhbw  mm1,mm7              ; unpack G1 to the higher bytes of
                                ; the 4 words of mm1
lea        edi, dRed            ; Load offset address of dRed.
psllw      mm3, 2               ; mulitiply each dR in 4 words of mm3 by 4
paddsw     mm2, mm0             ; 4 words in mm2 becomes R1, R1+dR,
                                ; R1+2dR, R1+3dR
punpckhbw  mm6,mm6              ; unpack B1 to the first 4 bytes of mm6
movq       [edi], mm3           ; store 4 dR to consecutive memory
                                ; location: dRed[4]
movq       mm0, mm4             ; load dG to 4 words of mm0
pmullw     mm0, [ecx]           ; multiplied by [3,2,1,0], mm0 becomes
                                ; [3dG,2dG,dG,0]
psllw      mm4, 2               ; mulitiply each dG in 4 words of mm4 by 4
mov        ebx, Rgbs            ; Load address of *Rgbs for storing
                                   ;intensities
movq       mm7, mm5             ; load dB to 4 words of mm7
mov        eax, 0ff00ff00h
paddsw     mm1, mm0             ; 4 words in mm2 becomes G1, G1+dG,
                                ; G1+2dG, G1+3dG
pmullw     mm7, [ecx]           ; multiplied by [3,2,1,0], mm7
                                ; becomes [3dB,2dB,dB,0]
pxor       mm0,mm0              ; clear mm0
movq       [edi - 8], mm4       ; store 4 dG to consecutive memory
                                ; location: dGreen[4]
psllw      mm5, 2               ; mulitiply each dB in 4 words of mm5 by 4
xor        ecx, ecx             ; clear ecx as a counter of j
punpcklbw  mm0,mm6              ; unpack B1 to the higher bytes of the
                                ; 4 words of mm0
movq       [edi - 16], mm5      ; store 4 dB to consecutive memory
                                ; location: dBlue[4]
paddsw     mm0, mm7             ; 4 words in mm0 becomes B1, B1+dB,
                                ; B1+2dB, B1+3dB
movdt      mm7, eax             ; load 0ff00ff00h to the first dword of mm7
movq       mm3,mm2              ; load R1,R2,R3,R4 to 4 words of mm3
xor        eax, eax             ; clear eax
movq       mm4,mm1              ; load G1,G2,G3,G4 to 4 words of mm4
mov        ax, numP             ; load numP to eax
punpckldq  mm7,mm7              ; mm7(0ff00ff00ff00ff00h) will serve as
                                ; a mask in J loop
;For each pass through of loopJ, RGB intensities of four
;pixels will be calculated and stored to the destination
;memory location of Rgbs[]. If numP is not a multitple of
;4, there will be up to 3 extra intensities assignment in
;the last pass of the loop. Caller should allocate enough
;memory for holding the extra intensities.
```

```
loopJ:
pand        mm4, mm7          ; clear the fractional part of
                              ; G1,G2,G3,G4 in mm4
psrlw       mm3, 8            ; shift the integer parts of R1,R2,R3,R4
                              ; to lower byte.
movq        mm5,mm0           ; copy B1,B2,B3,B4 into 4 words of mm5
por         mm3,mm4           ; combine G,R and load G1R1, G2R2,
                              ; G3R3, G4R4 to 4 ;words of mm3
psrlw       mm5, 8            ; shift the integer parts of
                              ; B1,B2,B3,B4 to lower bytes
movq        mm6,mm3           ; copy G1R1,G2R2,G3R3,G4R4 to mm6.
paddw       mm2, [edi]        ; add (4dR,4dR,4dR,4dR) to
                              ; (R1,R2,R3,R4) in mm2.
punpcklwd   mm3,mm5           ; unpack B1G1R1,B2G2R2 to 2 dwords of mm3
paddw       mm1, [edi - 8]    ; add (4dG,4dG,4dG,4dG) to
                              ; (G1,G2,G3,G4) in mm1
punpckhwd   mm6,mm5           ; unpack B3G3R3, B4G4R4 to 2 dwords of mm6
paddw       mm0, [edi - 16]   ; add (4dB,4dB,4dB,4dB) to
                              ; (B1,B2,B3,B4) in mm0
movq        mm4,mm1           ; copy G1,G2,G3,G4 into 4 words of mm4
movq        [ebx +4*ecx],mm3  ; store two B1G1R1, B2G2R2 to
                              ; memory location
movq        mm3,mm2           ; copy R1,R2,R3,R4 into 4 words of mm3
movq 8    [ebx +4*ecx], mm6   ; store B3G3R3, B4G4R4 to memory
                              ; location
add         ecx, 4            ; increase the J-counter by 4.
cmp         ecx, eax          ; compare counter ecx with num_Pjl
                              ; if ecx < numP, continue for next four
                                ;pixels
emms                          ; clear floating point stack
ret
Scan_ln_RGB_MMX EndP
END
```

II. Apéndice B. Índice de Tablas y Figuras