



Using MMX™ Instructions to Implement the Gouraud Shading Algorithm

Information for Developers and ISVs

From Intel® Developer Services
www.intel.com/IDS

March 1996

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

Note: Please be aware that the software programming article below is posted as a public service and may no longer be supported by Intel.

Copyright © Intel Corporation 2004

* Other names and brands may be claimed as the property of others.

Using MMX™ Instructions to Implement the Gouraud Shading Algorithm

March 1996

CONTENTS

1.0. INTRODUCTION

2.0 THE GOURAUD SHADING ALGORITHM

3.0. INPUT AND OUTPUT DATA REPRESENTATION

4.0. CODE PARTITIONING

 4.1. Flow in Data Setup

 4.2. Flow in Inner Loop Section

5.0. MMX™ TECHNOLOGY PERFORMANCE

APPENDIX A

1.0. 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™ 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.

2.0 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 1. Bilinear Interpolation

Ia
Is Ip Ie
Ic
Ib

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 2. Bilinear Interpolation Formulas

$$I_s = v * I_a + (1 - v) * I_b \quad 0 \leq v \leq 1 \quad (1)$$

$$I_e = t * I_a + (1 - t) * I_c \quad 0 \leq t \leq 1 \quad (2)$$

$$I_p = u * I_s + (1 - u) * I_e \quad 0 \leq u \leq 1 \quad (3)$$

$$I_{p(i+1)} = I_{p(i)} + dI \quad i = 1, 2, \dots, n-1 \quad (4)$$

$$dI = (I_e - I_s) / (n-1) \quad (5)$$

The bilinear interpolation is performed in two steps:

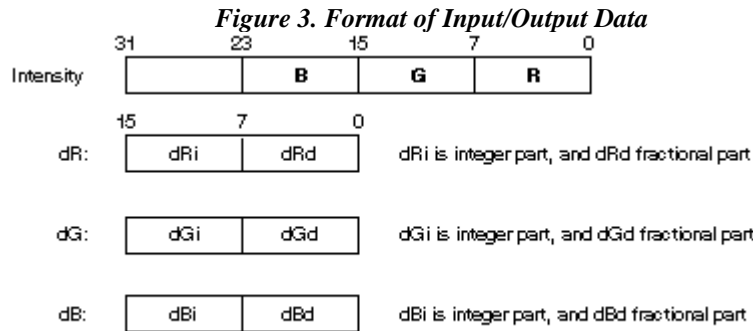
1. Intensities *I_s*, *I_e* at the pixels *S* and *E* are determined by linearly interpolating the intensities of the triangle vertices. This is shown in Formulas 1 and 2.
2. The intensity *I_p* at a pixel *P* on the scan line is also obtained by linearly interpolating along the scan line between *S* 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, *dI* is the incremental intensity for each successive pixel. Formula 5 shows how this is calculated.

3.0. 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.



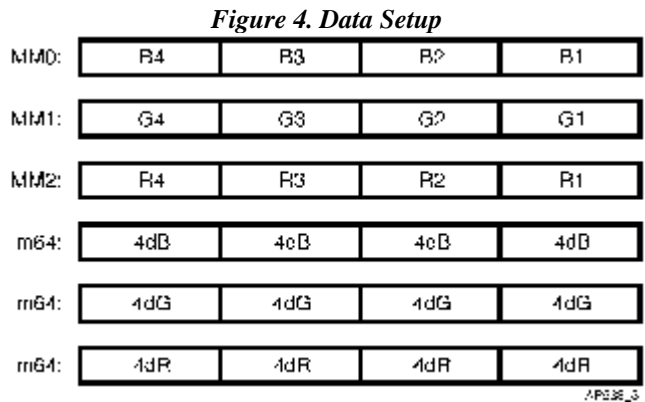
We assume that the ranges of RGB values are from 0 to 255 and that there are at least four pixels on the scan line. Hence the ranges of dR , dG and dB values are from $-255/4$ to $255/4$. The resulting intensities are stored in an array of DWORD. If there are fewer than four pixels on the scan line,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

4.0. CODE PARTITIONING

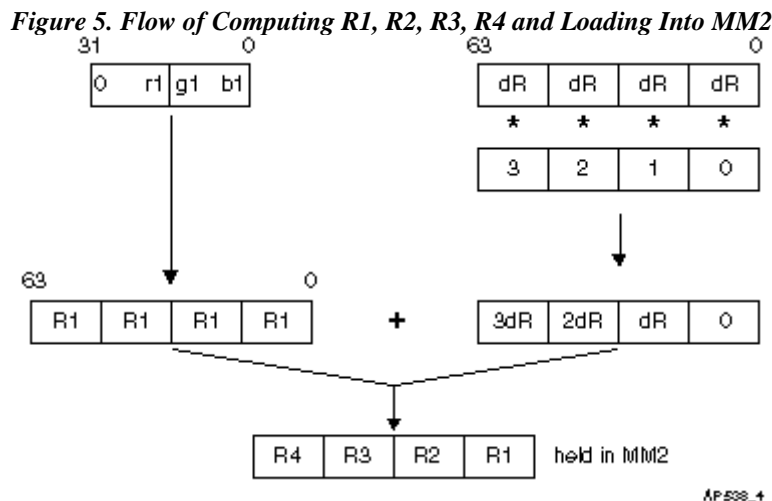
The procedure primarily consists of two main sections: a data setup section and an inner loop section in which intensities are assigned to four pixels. In both sections, the code fully exploits SIMD techniques to process four intensities in parallel.

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™ registers or memory locations.



The MMX registers MM2, MM1 and MM0 hold RGB values for four consecutive pixels, where $R2 = R1 + dR, \dots, R4 = R3 + dR \dots$ 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.



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.

Using MMX™ Instructions to Implement the Gouraud Shading Algorithm

March 1996

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 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 Appendix A.)

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

Using MMX™ Instructions to Implement the Gouraud Shading Algorithm

March 1996

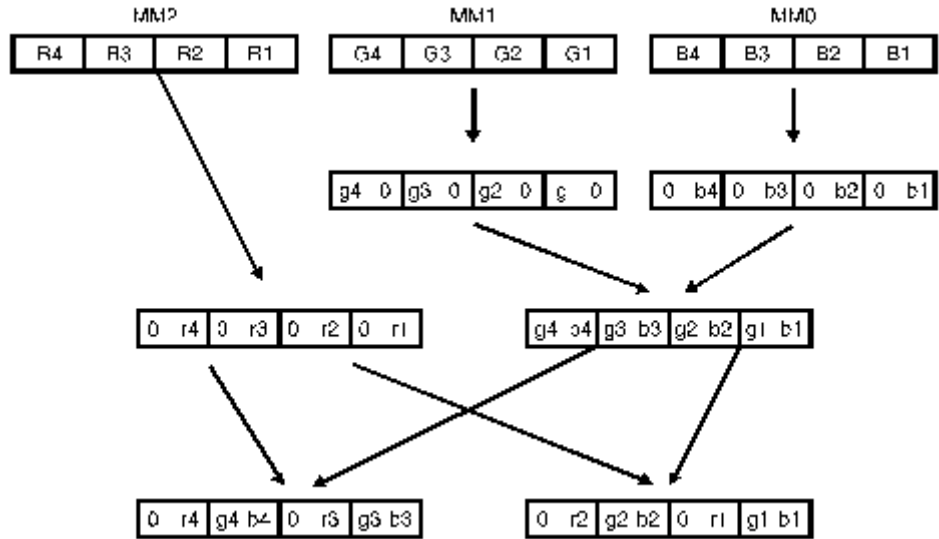
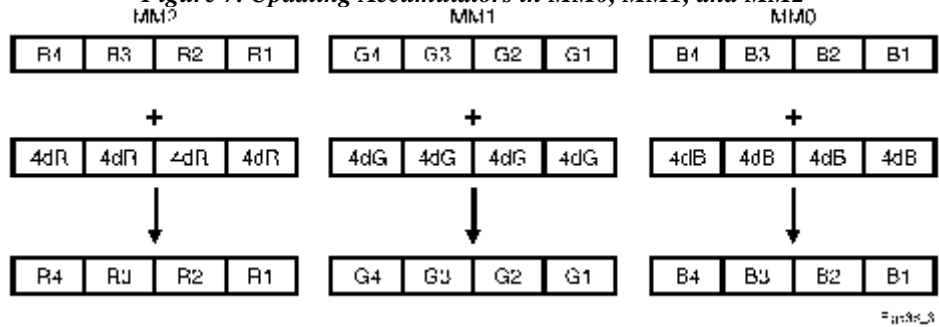


Figure 7. Updating Accumulators in MM0, MM1, and MM2



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 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
    
```

5.0. 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 to 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 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

Using MMX™ Instructions to Implement the Gouraud Shading Algorithm

March 1996

APPENDIX A

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
```

Using MMX™ Instructions to Implement the Gouraud Shading Algorithm

March 1996

```

; 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            ; multiply 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            ; multiply each dG in 4 words of mm4 by 4
mov      ebx, Rgbs        ; Load address of *Rgbs for storing intensities
```

Using MMX™ Instructions to Implement the Gouraud Shading Algorithm

March 1996

```
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        ; multiply 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 multiple 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
```

Using MMX™ Instructions to Implement the Gouraud Shading Algorithm

March 1996

```
emms                                     ; if ecx < numP, continue for next four pixels
ret                                     ; clear floating point stack
Scan_ln_RGB_MMX EndP
END
```