CONTENTS

1.0. INTRODUCTION
2.0. BILINEAR INTERPOLATION
3.0. THE MMX-BILINEAR INTERPOLATION PROCEDURE
   3.1. Format of the Procedure Parameters
   3.2. The Bilinear Interpolation Algorithm
APPENDIX A. OPTIMIZED BILINEAR INTERPOLATION ALGORITHM
APPENDIX B. SCALAR VERSION OF THE BILINEAR INTERPOLATION ALGORITHM
APPENDIX C. BILINEAR INTERPOLATION ALGORITHM IN C LANGUAGE SOURCE CODE
1.0 INTRODUCTION

The media extension to the Intel Architecture (IA) instruction set includes single-instruction multiple-data (SIMD) instructions. This application note presents examples of code that use the new MMX instructions, PMULH, PSRLD and PADDW to perform a bilinear interpolation of RGB values. The performance improvement over traditional IA code is primarily due to the significantly faster MMX technology multiply instructions. These new instructions are faster because MMX technology allows multiple 16-bit multiplies in parallel. While the IA multiply instruction IMUL) takes 11 cycles for a 16-bit multiply on a Pentium® processor, the MMX technology multiply instruction (PMULH) can perform four 16-bit multiplies with a throughput of only one cycle, with a three cycle latency.
2.0 BILINEAR INTERPOLATION

A common technique used for 3D rendering is to decompose the surface of objects into a large number of nearly planar triangles or rectangles. Their position in three-dimensional space is then mapped to the 2D display surface, and the individual triangles or rectangles drawn one pixel at a time. One technique for increasing the realism of the drawn image is to copy these triangles or rectangles from a bitmap image. These bitmaps are often called texture maps, since they are commonly used to represent texture-rich images such as wood grains.

This application note discusses one aspect of this process: accurately determining the color to display at a single pixel on the display surface. Prior to calling this code, it is assumed that the 3D software has calculated the position within the texture map that corresponds to the pixel to be drawn. In this application note, we assume that this position is calculated with fixed-point arithmetic (as opposed to floating-point). For information about using MMX instructions to implement this calculation, see Using MMX™ Instructions to Implement Bilinear Interpolation of Video RGB Values, Application Note AP-541, (Order Number 243026). The texture map position of the source pixel is traditionally described in \( (u,v) \) coordinates, where the integer part of \( (u,v) \) denotes a pixel that is physically stored in the texture bitmap. The fractional parts of \( u \) and \( v \) represent displacements to positions in the texture in between pixels that are physically stored in the texture bitmap (see Figure 1).

![Figure 1. Bilinear Interpolation of RGB Color at Pixel \((u,v)\)](image)

Bilinear interpolation uses a simple formula to estimate the color that would have been at the computed \( (u,v) \) coordinates if the texture map had been stored at a higher spatial resolution. Thus, bilinear interpolation can be interpreted as a time/space tradeoff. It allows the application to use smaller texture bitmaps, thus reducing storage space, at the cost of some extra computation time to do the bilinear interpolation. With MMX technology optimization, the cost of the computation time to perform the bilinear interpolation is significantly reduced. The bilinear interpolation formula is:

\[
R_{\text{result}} = R1 \times (1 - dU) \times (1 - dV) + \\
R2 \times (dU) \times (1 - dV) + \\
\frac{\text{U}}{\text{delta}} \times (dV) + \\
\frac{\text{V}}{\text{delta}} \times (dU) + \\
\frac{\text{delta}}{\text{delta}} \\
\]

\( R_{\text{result}} \) is the resulting RGB value, \( R1 \) and \( R2 \) are the RGB values of the two nearest pixels, \( dU \) and \( dV \) are the distances from the pixel to the nearest pixel in the texture map.
Using MMX™ Instructions to Implement Bilinear Interpolation of Video RGB Values
March 1996

R3 * (1 - dU) * (dV) +
R4 * (dU) * (dV)
G\_result = G1 * (1 - dU) * (1 - dV) +
G2 * (dU) * (1 - dV) +
G3 * (1 - dU) * (dV) +
G4 * (dU) * (dV)
B\_result = B1 * (1 - dU) * (1 - dV) +
B2 * (dU) * (1 - dV) +
B3 * (1 - dU) * (dV) +
B4 * (dU) * (dV)
3.0. THE MMX-BILINEAR INTERPOLATION PROCEDURE

This section describes the MMX-BilinearInterpolate procedure.

3.1. Format of the Procedure Parameters

The parameters of the MMX-BilinearInterpolate procedure are: a pointer to a texture map, TextureMap; a pointer to a color lookup table, ColorLookupTable; combined $u/u$ value; combined $v/v$ values; and a pointer to the return RGB value.

The TextureMap array is a two-dimensional array of 8-bit indexes into the ColorLookupTable. MMX-BilinearInterpolate expects the TextureMap array to be padded to 128 in the $y$ dimension. When this padding is done, address calculations involving two-dimensional indexing into TextureMap can be performed by a simple left shift by 7 rather than by a multiply. For convenience, the example in this application note hard codes this value. A few simple changes to the code would allow this value to be passed as a parameter.

The procedure also expects that the RGB values in the ColorLookupTable have been formatted as a quadword; where R, G, and B, values have been placed into the upper 8 bits of the 16-bit field. The range of RGB values is from binary 0 to 0xff00. Since the MMX technology PMULH instruction operates on signed numbers, these values are expected to be in the binary range 0 to 0x7f10. Typically, color lookup tables are small and rarely initialized, so preformatting them in this fashion as part of an application's outer loop is not expensive and greatly improves the efficiency of the bilinear interpolation procedure.

The parameters $u/u$ and $v/v$ should each be formatted as an unsigned long-word, where the upper 10 bits is the $u$ (or $v$) value, and the lower 22 bits is the $u$ (or $v$) value. This is a typical format, because it allows efficient calculation of the position of the source pixel in the texture map, and supports texture maps up to 1024 pixels wide (a practical upper limit). The procedure extracts the $u$ and $v$ values by extending the 10-bit $u$ and $v$ values to 16 bits. The delta values are extracted by keeping the 16 most significant bits, resulting in unsigned binary values in the binary range of 0 to 0xffff. However, as discussed above, the MMX technology PMULH instruction operates on signed numbers rather than unsigned numbers. After extracting the values from the 32-bit long-word into 16-bit quantities, the procedure scales them to a binary range of 0 to 7fff.

3.2. The Bilinear Interpolation Algorithm

There are three MMX technology multiply instructions, PMADD, PMULH, and PMULL. The PMADD instruction results in a 32-bit quantity, the PMULH and PMULL high each result in 16-bit quantities, but they keep only the upper or lower 16 bits respectively of the actual 32-bit result. Examination of the possible range of the input values to the procedure showed that using the PMULL instruction and keeping the upper 16 bits of a multiply result kept enough precision to produce correct final RGB results, as long as the color lookup table's RGB values were formatted to exist (scaled) in the upper 9 bits of the 16-bit RGB format (as discussed above).

The MMX technology multiply instructions perform multiplies on 16-bit data. The packed multiply and add instruction (PMADD) performs a multiply/accumulate operation by multiplying 16 by 16 bits into a 32-bit result that is added into a 32-bit accumulated value. The packed multiply high and packed high low
Using MMX™ Instructions to Implement Bilinear Interpolation of Video RGB Values

March 1996

instructions (PMULH and PMULL respectively) multiply 16 by 16 bits into a 32-bit result, but keep only 16 bits; PMULH keeps only the high order 16 bits, and PMULL keeps only the low order bits.

The MMX technology multiply instructions all multiply each 16-bit word in an MMX register in parallel. However, the PMADD instruction results in 32-bit double words, which allows only two multiplies in parallel. The PMULH and PMULL instructions result in 16-bit words, so when using one of these instructions it is possible to perform up to four multiplies in parallel. An example of register packing used in the MMX-BilinearInterpolate procedure is shown in Figure 2.

![Figure 2: MMX™ Register Packing](image)

The (1-u) term should be calculated as (0x8000 - u), but in order for the (1 - u) values to always be positive, (1-u) and (1-v) are calculated as (0x7fff - u). The scaling and multiply flow for the (1-u) * (1-v) term is shown in Figure 3. Using register packing and multiply flow allows the R, G, and B terms to be calculated in two multiplies, meaning that the entire bilinear interpolation formula takes only eight multiplies. In contrast, using the IA IMUL instruction to perform the bilinear interpolation requires 24 multiplies. Note that the resulting RGB remain in the lower 8 bits of their respective 16-bit fields to simplify pixel display by the calling procedure.
3.2. Optimizing the Bilinear Interpolation Algorithm

An optimized version of the MMX-BilinearInterpolate algorithm is shown in Appendix A, and a scalar version is shown in Appendix B. Two major optimization issues in the MMX-BilinearInterpolate procedure were instruction pairing and register dependence, and the three cycle latency associated with the result of a PMULH instruction (if the result of a PMULH is used before three cycles, cycles are inserted while the processor waits for the result. We advise scheduling instructions that don’t depend on the result of the PMULH during these otherwise lost cycles). The fact that MMX technology moves to and from memory can only be paired with MMX instructions that are register/register instructions, and
that PMULH can only be paired with scalar instructions, causes some minor pairing issues. Consequently, some minor recoding was done to allow for greater instruction pairing.

The scalar version of the MMX-BilinearInterpolate (see Appendix B) took 95 cycles to complete. After optimization (see Appendix A), the procedure took 66 cycles to complete. This optimization is due almost entirely to careful instruction pairing and instruction ordering to avoid the three cycle penalty associated with the PMULH instruction. For comparison, Appendix C has a C language source code version that operates using 32-bit unsigned multiplies where possible. Using MSVC 2.2, global optimization, and optimization for speed, this C language version was compiled and measured to take 259 clocks to complete. Even accounting for the fact that the C language source might itself be optimized to allow for better code generation, the version of the MMX-BilinearInterpolate procedure will probably be over 100 percent faster. As mentioned above, most of this improvement is due to the fact that in the MMX-BilinearInterpolate algorithm, eight PMULH multiplies at effectively one cycle each are performed to obtain a result, in comparison to the 24 IMUL multiplies taking 10 cycles each that are performed in the code.
APPENDIX A: OPTIMIZED BILINEAR INTERPOLATION ALGORITHM

; 'C' function prototype:
;
; void MMx_BiLinearInterpolate (BYTE TextureMap[][128],
; RGBQWORD ColorLookupTable[],
; DWORD dwUVal,
; DWORD dwVVal,
; LPRGBQWORD lpRGBOut)
;
; Abstract: this routine determines the bilinear interpolation of a RGB
; value at (U,V) in a transformed space, given a pointer to the
; original texture map, and a color lookup table for that texture
; map.
;
; the texture map was converted for efficiency, from a
; [58][72] byte array, to a [58][128] byte array with the
; elements from [j][72] to [j][127] unused.
;
; the color lookup table is a 256 element array, where each
; element is a RGBQWORD (see below).
;
; the U and V elements are formatted as 32-bit numbers where
; the upper 10 bits are the integer portion, and the lower
; 22 bits are the fractional portion of the (U,V) point whose
; color is being interpolated.
;
; NOTE: the fractional delta values are scaled
; from 0 - 0xffff.
; scale them down
; to 0 - 0x7fff
; to avoid signed/unsigned multiply issues
;
; the RGBQWORD is defined in 'C' as:
;
; typedef struct {
; WORD wB;
; WORD wG;
; WORD wR;
; WORD wUnused;
; } RGBQWORD, FAR *LPRGBQWORD;
;
; the values in the ColorLookupTable have been pre-formatted so that the
; RGB
; values are in bits 7-15.
;
; The interpolation can be depicted as:
Using MMX™ Instructions to Implement Bilinear Interpolation of Video RGB Values

March 1996

;            |           |          |
;            |           |          |
;            |           |          |
;            |           |          |
;            |           |          |
;            |           |          |
; RGB 1 | RGB 2 |          |
;            |           |          |
;            |           |          |
;            |           |          |
;            |           |          |
; RGB 3 | RGB 4 |          |
;            |           |          |
;            |           |          |

\[ R_{result} = R_1 \times (1 - \delta U) \times (1 - \delta V) + \]
\[ R_2 \times (\delta U) \times (1 - \delta V) + \]
\[ R_3 \times (1 - \delta U) \times (\delta V) + \]
\[ R_4 \times (\delta U) \times (\delta V) \]

\[ G_{result} = G_1 \times (1 - \delta U) \times (1 - \delta V) + \]
\[ G_2 \times (\delta U) \times (1 - \delta V) + \]
\[ G_3 \times (1 - \delta U) \times (\delta V) + \]
\[ G_4 \times (\delta U) \times (\delta V) \]

\[ B_{result} = B_1 \times (1 - \delta U) \times (1 - \delta V) + \]
\[ B_2 \times (\delta U) \times (1 - \delta V) + \]
\[ B_3 \times (1 - \delta U) \times (\delta V) + \]
\[ B_4 \times (\delta U) \times (\delta V) \]

the computations are performed in 15 bits, with the multiplies resulting in 31/30-bits, with 16-bit precision being recovered at the final result. Note: the shifts are 'performed' by pmulh.

30 bits  15 bits  15 bits
result = (1 - \delta U) \times (1 - \delta V)

then,

30 bits  15 bits
result = result >> 15

then,

31 bits  16 bits  15 bits
value = R_1 \times result

then,

31 bits  16 bits
value = value >> 15

NOTE: we leave the results in the lower 8 bits to simplify the pixel display
Using MMX™ Instructions to Implement Bilinear Interpolation of Video RGB Values

March 1996

;        by calling procedure.
;
;Environment:
;          MASM 6.11D
;
;***************************************************************************
;FUNCTION LIST:
;     (functions contained in this source module)
;
;    MMX_BiLinearInterpolate()
;
;***************************************************************************

.586
.MODEL FLAT, C
include iammx.inc
TAG     STRUCT 2t
  wB      WORD        ?
  wG      WORD        ?
  wR      WORD        ?
  wUnused WORD        ?
TAG     ENDS
RGBQWORD        TYPEDEF     TAG
LPRGBQWORD      TYPEDEF     FAR PTR TAG
.CODE
;-----------------------------------
; location of local vars on the stack
SAVEESP          EQU DWORD PTR [esp+0]
RGB1             EQU DWORD PTR [esp+8]
RGB2             EQU DWORD PTR [esp+16]
RGB3             EQU DWORD PTR [esp+24]
RGB4             EQU DWORD PTR [esp+32]
QTEMP            EQU DWORD PTR [esp+40]
QTEMP1           EQU DWORD PTR [esp+48]
ENDOFLOCALS      EQU DWORD PTR [esp+56]
LocalFrameSize   = 56
;***************************************************************************

MMx_BiLinearInterpolate PROC PUBLIC,
TextureMap      : PTR DWORD,
ColorLookupTable : PTR DWORD,
dwUVVal         : DWORD,
dwVVal          : DWORD,
lpRGBOut        : PTR DWORD
push     ; THE ASSEMBLER actually generates this
 mov     ebp, esp ; THE ASSEMBLER actually generates this
 sub     esp, LocalFrameSize
mov     SAVEESP, ebp ;save original esp to restore in epilgue
push    edx
and     esp, 0ffffff0h ;8-byte align start of local stack frame
mov     eax, dwUVVal ; get dwUVVal from parameter list.
mov     edx, dwVVal ; get dwVVal from parameter list.
mov     esi, eax ; save dwUVVal.
push    ebx
mov     ecx, edx ; save dwVVal.
shr     eax, 22 ; get integer portion of dwUVVal

}
Using MMX™ Instructions to Implement Bilinear Interpolation of Video RGB Values

March 1996

push esi
push ecx
and esi, 3fffffH
shr edx, 22 ; get integer portion of dwVVal
and eax, 0000ffffH ; eax = 0, U
shr esi, 7 ; shift by 7 instead of 6 to provide the 0-

0x7fff scaling
and ecx, 3fffffH
shr ecx, 7 ; shift by 7 instead of 6 to provide the 0-

0x7fff scaling
mov ebx, esi ; ebx = 0, dU
shl edx, 7 ; edx=V*128; 128 is hardcoded pitch value
push edi
shl ebx, 16 ; ebx = dU, 0
and edx, 0000ffffH ; edx = 0, V
or ebx, esi ; ebx = dU, dU
mov edi, ecx ; edi = 0, dV
shl edi, 16 ; edi = dV, 0
mov QTEMP, ebx ;
and esi, 0000ffffH ; esi = 0, dU
or edi, ecx ; edi = dV, dV
and ecx, 0000ffffH
mov QTEMP+4, ebx
mov QTEMP1, edi
mov QTEMP1+4, edi
mov mm4, QTEMP ; mm4 = dU | dU || dU | dU
mov mm5, QTEMP1 ; mm5 = dV | dV || dV | dV
mov mm1, mm4 ; mm1 = dU | dU || dU | dU
mov edi, 7fffH
mov ebx, 7fffH
sub edi, ecx ; edi = 0, 1-dV
sub ebx, esi ; ebx = 0, 1-dU
mov ecx, edi ; ecx = 0, 1-dV
mov esi, ebx ; esi = 0, 1-dU
shl ecx, 16 ; ecx = 1-dV, 0
add edx, eax ; edx = U + V*128
shl esi, 16 ; esi = 1-dU, 0
add edx, TextureMap ; edx = &TextureMap[U+V*128]
or edi, ecx ; edi = l-dV, l-dV
or ebx, esi ; ebx = l-dU, l-dU
mov QTEMP, edi ;
mov ci, BYTE PTR [edx]; bl = TextureMap[U+V*128]
mov eax, ColorLookupTable ; &ColorLookupTable[0]
and ecx, 000000ffH
mov QTEMP1, ebx ;
mov QTEMP1+4, ebx ;
mov mm6, QTEMP1 ; mm6 = 1-dU | 1-dU || l-dU | 1-dU
mov mm3, mm4 ; mm3 = dU | dU || dU | dU
mov QTEMP+4, edi ;
lea ecx, [eax+ecx*8] ; ebx = &ColorLookupTable[bl]
mov mm7, QTEMP ; mm7 = l-dV | l-dV || l-dV | l-dV
mov mm0, mm6 ; mm0 = 1-dU | 1-dU || 1-dU | 1-dU
pmulhw mm1, mm7 ; compute (dU) * (1 - dV)
mov mm2, mm6 ; mm2 = 1-dU | l-dU || 1-dU | 1-dU
pmulhw mm0, mm7 ; compute (1 - dU) * (1 - dV)
mov bl, BYTE PTR [edx+1] ; bl = TextureMap[U+V*128+1]
pmulhw mm2, mm5 ; compute 1 - dV

and      ebx, 000000ffh
pmulhw  mm3, mm5       ; compute (dU) * (dV)
pop     edi            ;
movq    mm4, mm0        ; mm4 = (1-dU)(1-dV)
movq    mm5, mm1        ; mm5 = (dU)(1-dV)
movq    mm0, [ecx]      ; mm0 =  xxxx  |    R1    |    G1    |    B1
movq    mm6, mm2        ; mm6 = (1-dU)(dV)
lea     ebx, [eax+ebx*8] ; ebx = &ColorLookupTable[bl]
mov     cl, BYTE PTR [edx+128] ; bl = TextureMap[U+V*128+128]
;   NOTE: we leave the results in the lower 8 bits to simplify
; the pixel display
; by calling procedure.

pmulhw  mm0, mm4        ; mm0 =  xxxx  |    R1(1-dU)(1-dV)  |    G1(1-dU)(1-dV)  |    B1(1-dU)(1-dV)
and     ecx, 000000ffH
movq    mm1, [ebx]      ; mm1 =  xxxx  |    R2    |    G2    |    B2
movq    mm7, mm3        ; mm7 = (dU)(dV)
psrlw   mm0, 5          ;
pmulhw  mm1, mm5        ; mm1 =  xxxx  |    R2(dU)(1-dV)  |    G2(dU)(1-dV)  |    B2(dU)(1-dV)
lea     ecx, [eax+ecx*8] ; ebx = &ColorLookupTable[bl]
pop     esi
mov     bl, BYTE PTR [edx+128+1] ; bl = TextureMap[U+V*128+128+1]
movq    mm2, [ecx]      ; mm2 =  xxxx  |    R3    |    G3    |    B3
psrlw   mm0, 5          ;
pmulhw  mm2, mm6        ; mm2 =  xxxx  |    R3(1-dU)(dV)  |    G3(1-dU)(dV)  |    B3(1-dU)(dV)
and     ebx, 000000ffH
lea     ebx, [eax+ebx*8] ; ebx = &ColorLookupTable[bl]
psrlw   mm1, 5          ;
movq    mm3, [ebx]      ; mm3 =  xxxx  |    R4    |    G4    |    B4
paddw   mm0, mm1        ;
pmulhw  mm3, mm7        ; mm3 =  xxxx  |    R4(dU)(dV)  |    G4(dU)(dV)
psrlw   mm2, 5          ;
pop     edx              ;
pop     ecx              ;
paddw   mm2, mm0        ;
psrlw   mm3, 5          ;
mov     eax, lpRGBOut
paddw   mm2, mm3        ;
movq    [eax], mm2      ; clear the FP stack
emms                        ;
pop     ebx              ;
; leave                     ; THE ASSEMBLER actually generates this
ret 0                       ;
MMX_BiLinearInterpolate ENDP
END
APPENDIX B: SCALAR VERSION OF THE BILINEAR INTERPOLATION ALGORITHM

;'C' function prototype:

void MMx_BiLinearInterpolate (BYTE TextureMap[][128],
RGBQWORD ColorLookupTable[],
DWORD dwUVal,
DWORD dwVVal,
LPRGBQWORD lpRGBOut)

Abstract: This routine determines the bilinear interpolation of a RGB value at (U,V) in a transformed space, given a pointer to the original texture map, and a colorlookuptable for that texture map.

The texture map was converted for efficiency, from a [58][72] byte array, to a [58][128] byte array with the elements from [j][72] to [j][127] unused.

The color lookup table is a 256 element array, where each element is a RGBQWORD (see below).

The U and V elements are formatted as 32-bit numbers where the upper 10 bits are the integer portion, and the lower 22 bits are the fractional portion of the (U,V) point whose color is being interpolated.

NOTE: the fractional delta values are scaled from 0 - 0xffff. Scale them down to 0 - 0x7fff to avoid signed/unsigned multiply issues.

The RGBQWORD is defined in 'C' as:

typedef struct {
    WORD wB;
    WORD wG;
    WORD wR;
    WORD wUnused;
} RGBQWORD, FAR *LPRGBQWORD;

The values in the ColorLookupTable have been pre-formatted so that the RGB values are in bits 7-15.

The interpolation can be depicted as:

| | | |
|RGB 1| RGB 2|
Using MMX™ Instructions to Implement Bilinear Interpolation of Video RGB Values

March 1996

R_result = R1 * (1 - delta U) * (1 - delta V) +
          R2 * (delta U) * (1 - delta V) +
          R3 * (1 - delta U) * (delta V) +
          R4 * (delta U) * (delta V)

G_result = G1 * (1 - delta U) * (1 - delta V) +
          G2 * (delta U) * (1 - delta V) +
          G3 * (1 - delta U) * (delta V) +
          G4 * (delta U) * (delta V)

B_result = B1 * (1 - delta U) * (1 - delta V) +
          B2 * (delta U) * (1 - delta V) +
          B3 * (1 - delta U) * (delta V) +
          B4 * (delta U) * (delta V)

The computations are performed in 15 bits, with the multiplies resulting
in 31/30-bits, with 16-bit precision being recovered at the final result.
Note: the shifts are 'performed' by pmulh.

30 bits       15 bits       15 bits
result =   (1 - delta U) * (1 - delta V)
then,

30 bits       15 bits
result = result >> 15
then,

31 bits       16 bits       15 bits
value = R1      * result
then,

31 bits       16 bits
value = value >> 15

NOTE: We leave the results in the lower 8 bits to
simplify the pixel display by calling procedure.
Using MMX™ Instructions to Implement Bilinear Interpolation of Video RGB Values
March 1996

; Environment:
; MASM 6.11D
;
;******************************************************************************
; FUNCTION LIST:
; (functions contained in this source module)
;
; MMX_BiLinearInterpolate()
;
;******************************************************************************

.586
.MODEL FLAT, C
include iammx.inc
TAG          STRUCT 2t
  wB          WORD        ?
  wG          WORD        ?
  wR          WORD        ?
  wUnused     WORD        ?
TAG          ENDS
RGBQWORD      TYPEDEF     TAG
LPRGBQWORD    TYPEDEF     FAR PTR TAG

;-----------------------------------
; location of parameters on the stack
TextureMap    = 8
ColorLookupTable = 12
dwUVal       = 16
dwVVal       = 20
lpRGBOut     = 24
;-----------------------------------
; location of local vars on the stack
RGB1         = -8
RGB2         = -16
RGB3         = -24
RGB4         = -32
QTEMP        = -40
.CODE
;***************************************************************************
;
MMX_BiLinearInterpolate PROC PUBLIC
    push    ebp
    mov     ebp, esp
    sub     esp, 40             ; reserve space for locals.
    push    ebx
    push    ecx
    push    edx
    push    esi
    push    edi
;
    ; get fractional parts
    ; of U from the dwUVal parameter on
    ; the stack:
    mov     eax, dwUVal[ebp]     ; get dwUVal from parameter list.
    mov     esi, eax             ; save dwUVal.
    and     esi, 3ffffffH
Using MMX™ Instructions to Implement Bilinear Interpolation of Video RGB Values

March 1996

shr esi, 7 ; extract bottom 22 bits from dwUVal.
; (NOTE: we shift by 7 instead of 6
; to provide the 0-0x7fff scaling)
and esi, 0000ffffH ; esi = dU
shr eax, 22 ; get integer portion of dwUVal
and eax, 0000ffffH ; eax = U
mov edx, dwVVal[ebp] ; get dwVVal from parameter list.
mov ecx, edx ; save dwVVal.
and ecx, 3fffffH
shr ecx, 7 ; extract bottom 22 bits from dwVVal.
; (NOTE: we shift by 7 instead of 6
; to provide the 0-0x7fff scaling)
and ecx, 0000ffffH ; ecx = dV
shr edx, 22 ; get integer portion of dwVVal
and edx, 0000ffffH ; edx = V
shr edx, 7 ; multiply the V value by 128,
; to account for the pitch of
; TextureMap[][128] array.
; (edx = V*pitch.)

;-----------------------------------
; build the multiply quadwords
mov ebx, esi ; ebx = 0, dU
shl ebx, 16 ; ebx = dU, 0
or ebx, esi ; ebx = dU, dU
mov QTEMP[ebp], ebx ;
mov QTEMP[ebp+4], ebx ;
movq mm4, QTEMP[ebp] ; mm4 = dU | dU || dU | dU
mov ebx, 7fffH
and esi, 0000ffffH
sub ebx, esi ; ebx = 0, 1-dU
mov esi, ebx ; esi = 0, 1-dU
shl esi, 16 ; esi = 1-dU, 0
or ebx, esi ; ebx = 1-dU, 1-dU
mov QTEMP[ebp], ebx ;
mov QTEMP[ebp+4], ebx ;
movq mm1, mm4 ; mm1 = dU | dU || dU | dU
movq mm6, QTEMP[ebp] ; mm6 = 1-dU | 1-dU || 1-dU | 1-dU
mov ebx, ecx ; ebx = 0, dV
shl ebx, 16 ; ebx = dV, 0
or ebx, ecx ; ebx = dV, dV
mov QTEMP[ebp], ebx ;
mov QTEMP[ebp+4], ebx ;
movq mm5, QTEMP[ebp] ; mm5 = dV | dV || dV | dV
movq mm0, mm6 ; mm0 = 1-dU | 1-dU || 1-dU | 1-dU
mov ebx, 7fffH
and ecx, 0000ffffH
sub ebx, ecx ; ebx = 0, 1-dV
mov ecx, ebx ; ecx = 0, 1-dV
shl ecx, 16 ; ecx = 1-dV, 0
or ebx, ecx ; ebx = 1-dV, 1-dV
mov QTEMP[ebp], ebx ;
mov QTEMP[ebp+4], ebx ;
movq mm7, QTEMP[ebp] ; mm7 = 1-dV | 1-dV || 1-dV | 1-dV
movq mm3, mm4 ; mm3 = dU | dU || dU | dU
pmulhw mm0, mm7 ; compute (1 - dU) * (1 - dV)
pmulhw mm1, mm7 ; compute (dU) * (1 - dV)
movq mm2, mm6 ; mm2 = 1-dU | 1-dU || 1-dU | 1-dU
Using MMX™ Instructions to Implement Bilinear Interpolation of Video RGB Values

March 1996

using MMX™ instructions to implement bilinear interpolation of video rgb values

pmulhw mm3, mm5 ; compute (dU) * (dV)

pmulhw mm2, mm5 ; compute (1 - dU) * (dV)

movq mm4, mm0 ; mm4 = (1-dU)(1-dV)

movq mm5, mm1 ; mm5 = (dU)(1-dV)

movq mm6, mm2 ; mm6 = (1-dU)(dV)

movq mm7, mm3 ; mm7 = (dU)(dV)

add edx, eax ; edx = U + V*128

mov eax, ColorLookupTable[ebp] ; &ColorLookupTable[0]

add edx, TextureMap[ebp] ; edx = &TextureMap[U+V*128]

;-----------------------------------

; NOTE: We leave the results in the lower 8 bits to simplify

; the pixel display by calling procedure.

;-----------------------------------

; compute:

; R1 * (1 - dU) * (1 - dV)

; G1 * (1 - dU) * (1 - dV)

; B1 * (1 - dU) * (1 - dV)

xor ebx, ebx

mov bl, BYTE PTR [edx] ; bl = TextureMap[U+V*128]

lea ebx, [eax+ebx*8] ; ebx = &ColorLookupTable[bl]

movq mm0, [ebx] ; mm0 = R1(1-dU)(1-dV)

pmulhw mm0, mm4

psrlw mm0, 5

;-----------------------------------

; compute:

; R2 * (dU) * (1 - dV)

; G2 * (dU) * (1 - dV)

; B2 * (dU) * (1 - dV)

xor ebx, ebx

mov bl, BYTE PTR [edx+1] ; bl = TextureMap[U+V*128+1]

lea ebx, [eax+ebx*8] ; ebx = &ColorLookupTable[bl]

movq mm1, [ebx] ; mm1 = R2(dU)(1-dV)

pmulhw mm1, mm5

psrlw mm1, 5

;-----------------------------------

; compute:

; R3 * (1-dU) * (dv)

; G3 * (1-dU) * (dv)

; B3 * (1-dU) * (dv)

xor ebx, ebx

mov bl, BYTE PTR [edx+128] ; bl = TextureMap[U+V*128+128]

lea ebx, [eax+ebx*8] ; ebx = &ColorLookupTable[bl]

movq mm2, [ebx] ; mm2 = R3(1-dU)(dv)

pmulhw mm2, mm6

psrlw mm2, 5

;-----------------------------------

; compute:

; R4 * (dU) * (dv)

; G4 * (dU) * (dv)
Using MMX™ Instructions to Implement Bilinear Interpolation of Video RGB Values

March 1996

; B4 * (dU) * (dV)
xor ebx, ebx
mov b1, BYTE PTR [edx+128+1] ; b1 = TextureMap[U+V*128+128+1]
lea ebx, [eax+ebx*8] ; ebx = &ColorLookupTable[b1]
movq mm3, [ebx] ; mm3 = xxxx | R4 || G4
| B4
pmulhw mm3, mm7 ; mm3 = xxxx | R4(dU)(dV) ||

G4(dU)(dV) | B4(dU)(dV)
psrlw mm3, 5 ;

; now add'm up...
paddw mm0, mm1
paddw mm2, mm3
paddw mm0, mm2

; save the result out into lpRGBOut
mov ebx, lpRGBOut[ebp]
movq [ebx], mm0

alldone: ; restore stack
pop edi ;
pop esi ;
pop edx ;
pop ecx ;
pop ebx ;
mov esp, ebp
pop ebp ;
emms ; clear the FP stack
ret 0 ;

MMX_BiLinearInterpolate ENDP
END
APPENDIX C: BILINEAR INTERPOLATION ALGORITHM IN C LANGUAGE SOURCE CODE

Abstract: This routine determines the bilinear interpolation of a RGB value at (U,V) in a transformed space, given a pointer to the original texture map, and a colorlookup table for that texture map.

The texture map was converted for efficiency, from a [58][72] byte array, to a [58][128] byte array with the elements from [j][72] to [j][127] unused.

The color lookup table is a 256 element array, where each element is a RGBQWORD (see below).

The U and V elements are formatted as 32-bit numbers where the upper 10 bits are the integer portion, and the lower 22 bits are the fractional delta portion of the (U,V) point whose color is being interpolated.

NOTE: the fractional delta values are scaled from 0 - 0xffff.

scale them down to 0 - 0x7fff
to avoid signed/unsigned multiply issues.

(we do this in this 'c' code, although it is really only an issue for the MMX assembly version, since MMX doesn't support unsigned multiplies.

The RGBQWORD is defined as:

```
typedef struct {
    WORD wB;
    WORD wG;
    WORD wR;
    WORD wUnused;
} RGBQWORD, FAR *LPRGBQWORD;
```

The values in the ColorLookupTable have been pre-formatted so that the RGB values are in bits 7-15.

This routine returns as output, a RGBQWORD value into the LPRGBQWORD parameter.

The interpolation can be depicted as:

```
|          |          |          |
|          |          |          |
|          |          |          |
|          |          |          |
|               ^                       |
| (U,V)        | V          |          |
|               v                       |
|<-------->*    | delta      |          |
|                |            |          |
|                |            |          |
|                |            |          |
```

```
R_result = R1 * (1 - delta U) * (1 - delta V) +
R2 * (delta U) * (1 - delta V) +
R3 * (1 - delta U) * (delta V) +
```

Using MMX™ Instructions to Implement Bilinear Interpolation of Video RGB Values

March 1996

R4 * (delta U) * (delta V)

G_result = G1 * (1 - delta U) * (1 - delta V) +
G2 * (delta U) * (1 - delta V) +
G3 * (1 - delta U) * (delta V) +
G4 * (delta U) * (delta V)

B_result = B1 * (1 - delta U) * (1 - delta V) +
B2 * (delta U) * (1 - delta V) +
B3 * (1 - delta U) * (delta V) +
B4 * (delta U) * (delta V)

Environment: MSVC v2.2

FUNCTION LIST:
(functions contained in this source module)

****************************************************************************

#include <windows.h>
#include "bilin.h"

void
BilinearInterpolate (BYTE       TextureMap[][128],
RGBQWORD   ColorLookupTable[],
DWORD      dwUVal,
DWORD      dwVVal,
LPRGBQWORD lpRGBOut)
{
    DWORD dwUDelta = (dwUVal & 0x003fffffl) >> 6;
    DWORD dwVDelta = (dwVVal & 0x003fffffl) >> 6;
    DWORD dwUInt   = dwUVal >> 22;
    DWORD dwVInt   = dwVVal >> 22;
    RGBQWORD RGB1 = ColorLookupTable[TextureMap[dwVInt][dwUInt]];  
    RGBQWORD RGB2 = ColorLookupTable[TextureMap[dwVInt][dwUInt + 1]]; 
    RGBQWORD RGB3 = ColorLookupTable[TextureMap[dwVInt + 1][dwUInt]];  
    RGBQWORD RGB4 = ColorLookupTable[TextureMap[dwVInt + 1][dwUInt + 1]];  

    WORD w1, w2, w3, w4;
    DWORD dwTemp1, dwTemp2, dwTemp3, dwTemp4;
    dwTemp1 = (0x100000L - dwUDelta) * (0x100000L - dwVDelta);
    dwTemp2 = dwUDelta * (0x100000L - dwVDelta);
    dwTemp3 = (0x100000L - dwUDelta) * dwVDelta;
    dwTemp4 = dwUDelta * dwVDelta;
    dwTemp1 = dwTemp1 >> 16;
    dwTemp2 = dwTemp2 >> 16;
    dwTemp3 = dwTemp3 >> 16;
    dwTemp4 = dwTemp4 >> 16;
    // compute B
    w1 = RGB1.wB >> 7;
    w2 = RGB2.wB >> 7;
    w3 = RGB3.wB >> 7;
    w4 = RGB4.wB >> 7;
    lpRGBOut->wB = (WORD)(((w1 * dwTemp1) >> 16) +
                          ((w2 * dwTemp2) >> 16) +
                          ((w3 * dwTemp3) >> 16) +
                          ((w4 * dwTemp4) >> 16));

    // Note: We leave the results in the lower 8 bits to simplify
    // the pixel display by calling procedure.
    // compute G
    w1 = RGB1.wG >> 7;
    w2 = RGB2.wG >> 7;
    w3 = RGB3.wG >> 7;
w4 = RGB4.wG >> 7;
lpRGBOut->wG = (WORD)(((w1 * dwTemp1) >> 16) +
    ((w2 * dwTemp2) >> 16) +
    ((w3 * dwTemp3) >> 16) +
    ((w4 * dwTemp4) >> 16));
// Note: We leave the results in the lower 8 bits to
// simplify the pixel display by calling procedure.
// compute R
w1 = RGB1.wR >> 7;
w2 = RGB2.wR >> 7;
w3 = RGB3.wR >> 7;
w4 = RGB4.wR >> 7;
lpRGBOut->wR = (WORD)(((DWORD)w1 * dwTemp1) >> 16) +
    (((DWORD)w2 * dwTemp2) >> 16) +
    (((DWORD)w3 * dwTemp3) >> 16) +
    (((DWORD)w4 * dwTemp4) >> 16));
// Note: We leave the results in the lower 8 bits
// to simplify the pixel display by calling procedure.