MMX™ Technology for 3D Rendering

Information for Developers and ISVs

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

March 1996

MMX™ Technology for 3D Rendering

March 1996

CONTENTS

Executive Summary

Intel's MMX™ Technology processes multiple integer data items in parallel, 64 bits at a time. It can speed processing of pixels in 3D graphics, compared to straight Intel Architecture code which handles at most 32 bits at a time. Thus MMX technology may enable higher frame rates and/or higher quality images.

MMX technology best fits 16-bit or 24-bit color rendering, although it can assist 8-bit also. Intel and other developers have written 3D rendering code using MMX technology. Benefits were found in alpha blending, Gouraud shading, Z-Buffering, pixel-depth conversion, image copying, bilinear filtering, texture mapping, and procedural texturing. MMX technology algorithms can be 2x to 4x faster in 16 and 24-bit color; but 8-bit color algorithms typically speedup only 30% or less. These speedups come from:

- Parallel SIMD(Single Instruction Multiple Data) processing
- 8-byte memory accesses via the **MOVQ** instruction
- "Saturating" arithmetic: which ensures that brightening a white pixel (FF FF FF in 24-bit RGB) by adding a small amount (01 01 01 hex) results in another white, not a black (00 00 00) pixel.
- Mulitply-add instructions (**PMADD**) executing in a single-clock.
- Packed Compare (**PCMP**) operations, which allow removal of data-dependent branches from many algorithms.

For best performance, SW designers using MMX technology must carefully tune for system bandwidths. Existing PCI bus and graphics chips constrain throughput to graphics cards to about 80 MBytes/sec for writes, and far less for reads. Likewise current main-memory systems typically have less than 160 MBytes/sec total sustainable bandwidth.

However, the external cache of the CPU has more than twice this speed, and the internal cache more than 10x the main memory speed. Code designers should ensure their algorithms and data stay in the caches as much as possible, and should avoid reading data from the graphics card (via the PCI bus). Even the new A.G.P. bus is significantly faster for writes than for reads from the CPU, so software should avoid reading directly from the RAM on the graphics card. The A.G.P. bus facilitates sharing of main memory regions at high bandwidth by both the graphics chip and the CPU.

Creating 3D applications and content for MMX technology is very much like creating for 3D hardware accelerators, as both use 16-bit or 24-bit RGB formats. In some cases, software rendering runs faster, and certainly more flexibly, than current 3D hardware. In other cases, it makes sense to combine foreground objects and special effects, rendered in software, with hardware-rendered backgrounds. Hardware imposes certain overhead in setting-up registers and synchronization, but of course has higher bandwidth to graphics memory because both memory and the accelerator are on the same end of the PCI (or other) bus.

Target Audience

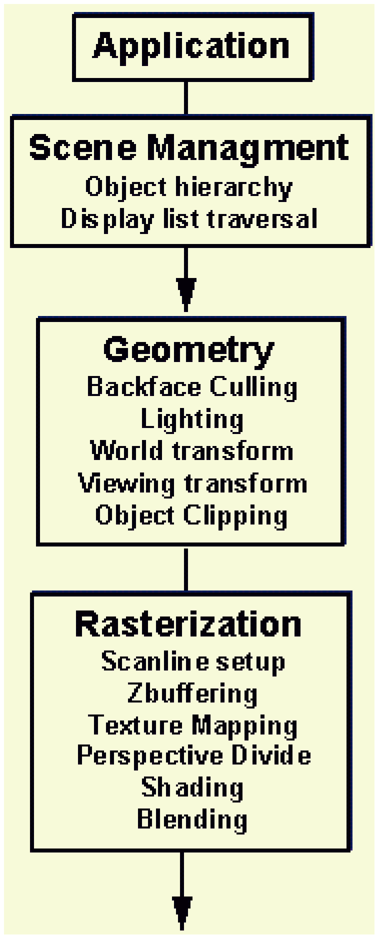
This paper was written for programmers and technology managers, as an overview of MMX technology for 3D graphics. It provides hints on the most beneficial tactics and the tradeoffs in implementations.

Rather than writing custom code, programmers may want to use an available 3D API library built on MMX technology, such as Microsoft's Direct3D*, Criterion's Renderware*, or Argonaut's BRender*. But this paper gives insight to rasterization in custom code. We assume the reader is familiar with 3D graphics terms and concepts.

Rendering Pipeline

The 3D graphics rendering pipeline, shown in Figure 1, is controlled by an application, usually through calls to an API. Scene management may be integrated in the app, or may be part of the 3D pipeline beneath the API. The 3D pipe processes a geometric database of points or vertices, handed to it by the scene manager. These vertices are linked into polygons, usually triangles. Typical scenes have 300 to 30k polygons, requiring 10k to 1M poly/sec processing for rendering 30 frames/sec. The final output is pixels, usually at rates of 5-100 Million per second. We limit the scope of our discussion to pixel processing, aka "rendering" or "rasterization". More information on geometry appears in the Appendix.

Figure 1: 3D Rendering Pipeline.



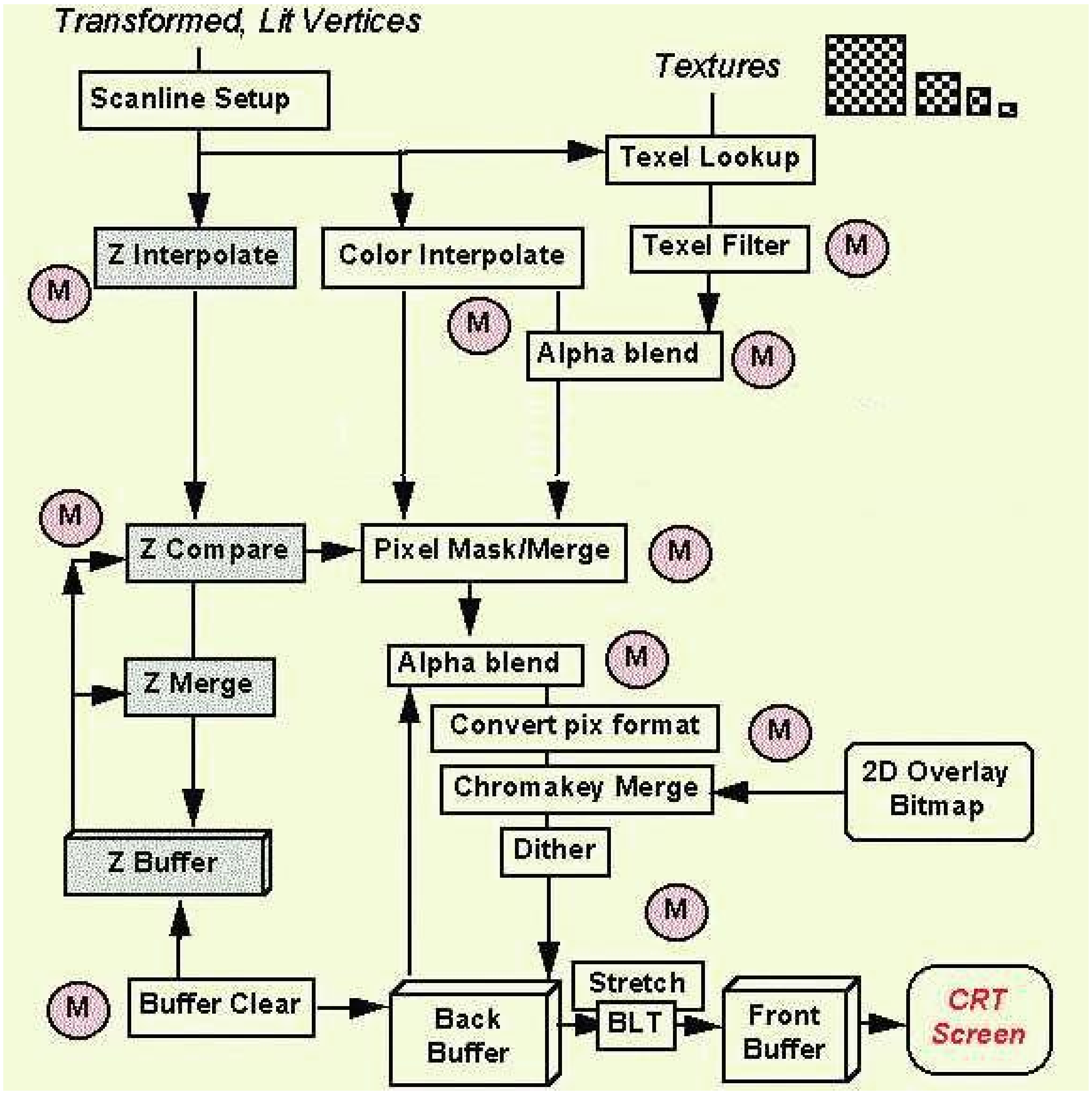
Rasterization

Figure 2 shows a typical 3D rasterization dataflow. For input, it receives transformed, lit vertices, and linking structures such as discrete triangles (three vertices each) or triangle meshes. In a mesh, each new vertex defines a new triangle, whose other two vertices are the preceeding two in the mesh list. For final output, the pipeline draws images on a CRT screen. The circled "M" symbols show areas where MMX technology can add quality or performance.

Each vertex consists of

- X, Y, and Z coordinates in "screen space"
- R, G, and B color components. Possibly "A" (alpha) transparency factor also.
- U and V texture coordinates (for polys which are to be textured)

Figure 2: 3D Rasterization. Note: the sequence of operations may vary.



Other features of vertices are typically maintained as a "state" variable of the rendering pipeline. These variables must be changed explicitly each time a new texture, shading, blending, or lighting model is desired. Typical state variables include:

- Texture -- a pointer to the current bitmap to be used for texturing, if any
- Texture Filtering Mode -- point sampling, bilinear filtering, MipMapped, etc...
- Texture Application Mode -- wrap, modulate, decal, blend, etc...
- Shading Mode -- flat, Gouraud (smooth), pseudo-specular, or Phong (highlighted)
- Material -- determines light reflectivity
- Z-buffering -- off or on, and type of Z-compare
- Clipping -- off or on, various types

- Blending mode -- none, source, destination, stipple (screen door), etc...
- Antialiasing -- off or on
- Dithering -- off or on
- Fog -- off or on, and variance with depth (Z)

For more thorough discussion of vertex information and states in a "typical" 3D library, refer to Microsoft's Direct3D* descriptions. In fact, the Direct3D rendering pipeline incorporates MMX technology in many of the steps discussed below, and readers are encouraged to experiment with Direct3D.

The sequence of operations on vertices can vary, but it usually follows a pattern like:

1. **Clip and Cull (Remove) Polygons** -- which are outside of the viewport or are backfacing, on the "backside" of the object. From partially-clipped objects, new polygons must be created which exclude the clipped-out areas.

2. **Setup Scan Lines** -- A scan line is a "horizontal" or constant-Y-value sequence of pixels. By calculating the slopes of the edges of each triangle, we determine the start and end pixels of each scan line, the number of pixels on the scanline, and the change in r, g, b, u, v, and Z per pixel. These deltas are often called dr, dg, db, du, dv, and dZ.

3. **Setup Texture Perspective Correction** -- For texturing, additional parameters for each scanline, or even smaller spans of scanlines, are required to correct for the foreshortening imposed by a perspective view. Without this correction factor, textures distort and show discontinuities along polygon edges (see Figure 3).

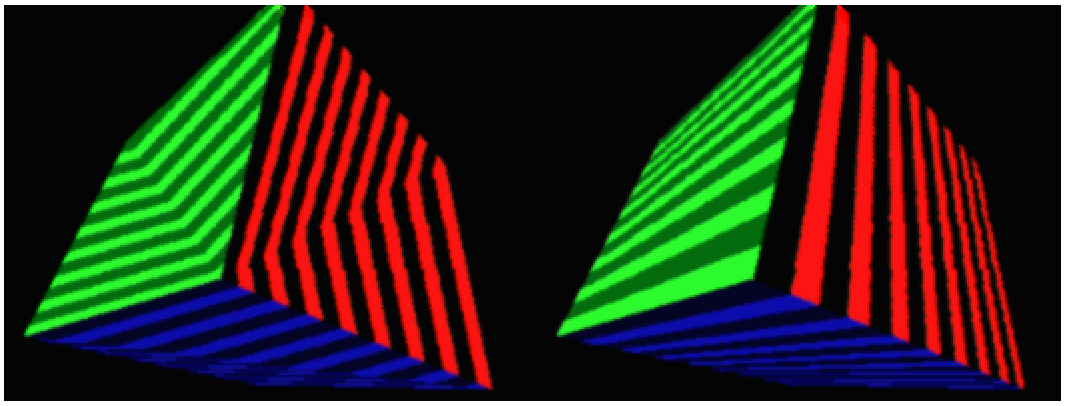


Figure 3: Perspective Correction Example. The striped cube on the left lacks correction, and shows discontinuties in textures at triangle edges.

Perfect perspective correction requires two divides PER PIXEL, or a divide and two muliplications. Perspective correction can be approximated without divides, by adding second-order (quadratic) differentials to the du & dv values at each pixel. These differentials are labeled ddu & ddv. See the app note on perspective-correction quadratic approximation. Alternatively, the algorithm can do the divide once per 8 or 16 pixels, and linearly interpolate for the intervening 7 or 15 pixels.

4. **Draw each pixel** on each scan line

While the above setup operations can benefit from the parallelism of MMX technology, the number of calculations is limited, and they often use floating point arithmetic. Much more benefit accrues in the subsequent steps of drawing each pixel on each scanline. To draw each pixel, a variety of operations occur, depending on the state variables of the renderer discussed earlier. The order of operations may vary, and may not include all of the following, but generally they are:

1. **Hidden Surface Removal**-- Determine whether the pixel is hidden by others in the scene nearer the viewer -- that is, if the Z-value (depth in the 3D coordinate system) of the pixel is greater than the one already drawn for a given X,Y location, don't draw it. This can be acheived by Z-buffering or by pre-sorting polygons or scan-line spans. For Z-buffering, the Z-values for pixels already drawn in the image must be refetched from the buffer, compared to new values, and conditionally updated. With MMX technology, a parallel compare and a packed AND*OR operation can merge the Z-values, as described in Intel's Z-buffer app note .

2. **Color**-- Calculate the pixel color via flat shading or Gouraud Shading, and/or texture mapping. MMX technology can Gouraud-shade two or four pixels concurrently, at rates of three to eight clocks per pixel. See the app notes Gouraud Shading Pixel Quads and Simplified Gouraud Shading .

3. **Texture**-- Look up the texture element(s), or texel(s) which are to be mapped to the current pixel. Multiply/add them together, if linear or bilinear filtering or trilinear filtering is specified. Texturing may require looking up a software-based palette for each texel, if the source bitmap is palettized (the usual case). It may also involve decompressing the texture via some statistical algorithm. And it may involve Gouraud shading to combine the underlying material color and/or light color with the texture.

   MMX instructions can also generate a without requiring a bitmap source for texture. Thus less memory bandwidth and space are consumed, and less aliasing or blockiness occurs for magnified views.

4. **Highlight** the pixel, by mulitplying-adding a value based on the current light & material, if specular shading is enabled. The multiply-add is very inexpensive with MMX technology.

5. **Blend** the resulting pixel with the pixel already in the image, if blending is enabled. Again the **PMADD** (Packed Multiply-Add) instruction speeds this considerably. MMX technology code for alpha blending can be found on Intel's Web site. Also haze and fog effects can be created by a similar mulitply-add blending operation.

6. **Convert** to final pixel format for display -- typically 24-bit in the above pipeline, but 16-bit on the screen. Again, there is an app note for an optimized 24-bit to 16-bit converter.

7. **Dither** the pixel color, if dithering is specified. Dithering is usually needed for 8-bit color, and sometimes for 16-bit. It allows a limited color set to approximate a broader range, by mixing

groups of varying-color pixels in a semi-random pattern. Without dithering, color gradients like sky or sunset tend to show "banding" artifacts.

8. **Copy**, or BLT (Bit-aligned BLock Transfer) the entire image from the backbuffer to the display buffer (frontbuffer). SW rendering is done to main memory or offscreen memory, not directly to the final display area. This is called double buffering, used because direct writes to the display are usually slower. Writes to the frontbuffer also cause artifacts due to the way the CRT beam periodically refreshes the screen. The copy or swapping of buffers is commonly accelerated by hardware, and is sometimes called "page flipping".

9. **Stretch** or zoom or replicate pixels, if specified. The stretch may occur during the BLT mentioned earlier. An appnote on 2x 8bit Image Scaling shows how. It would need adaptations for 16-bit or 24-bit images.

10. **Overlay** or merge 2D bitmaps with the 3D image, if specified. This overlaying puts sprites or effects atop the scene, via chromakeying -- pieces of the rectangular 2D bitmap which should not appear are usually pre-coded as a key color, and the renderer avoids writing pixels of this color to the result. Optimized code for sprite overlay has been written.

11. **Clear** the backbuffer and Z-buffer before drawing the next frame. For blasting values (like zeros or FFs) to memory, the MMX instruction **MOVQ** writes data at the maximum bandwidth of the CPU external bus (for example, 8 bytes every 3 clocks at 66 MHz, or 177 MB/sec in many Pentium(R) processor systems). Note that backbuffers and Z-buffers should be allocated in main memory for SW-based rendering; access to PCI memory from the CPU is much slower. Note also that various schemes exist to avoid clearing the entire buffers every frame. For example, the backbuffer may be partially cleared via "dirty rectangle" tracking. Also the Z-buffer may need to be cleared only every four frames, if software updates the Z-compare results with an offset specific to each of the four frames that makes each frame entirely closer to the camera than the preceeding.

Pixel Depth Tradeoff: 8 bit, 16 bit, 24 bit

MMX Technology does best on 16-bit hicolor and 24-bit true color data. The packed add, multiply, and logical operations actually make 24-bit the only format attractive for calculations, as three 8-bit R, G, and B components or 8.8 bit fixed point (48 bits total for RGB). After completing calculations (like Gouraud shading, alpha, etc...), the algorithm can convert to RGB16 (555 or 565) for updating the display buffer. MMX technology has no built-in arithmetic nor packing for 5-bit chunks.

24-bit internal operations allow high-quality features which are not possible with 8-bit palettized color, including:

- Alpha blending for transparency, fog, haze
- True RGB (multi-colored) lights for orange firelight, etc...
- Pseudo-specular highlighting
- More than 256 colors on the screen at once
- Truly smooth shading, free of banding artifacts
- Less (or no) dithering artifacts
- No palette flashes or clashes with other application palettes.
- Compatibility with hicolor content (textures) and features, similar to HW accelerators

Of course, the amount of graphics memory in the system determines whether the application can use 16-bit or 24-bit color. For 1MB graphics cards, the app probably may stay with 8-bit palettized (640 x 480 x 2bytes requires 600KB, which can fit in 1MB only if not double-buffered, or if the backbuffer is in system memory). For 2MB or 4MB, 16-bit is attractive, as the 1.2MB requirement of 640 x 480 double-buffered still leaves 800KB for texture caching and/or Z-buffering.

Eight-bit color does permit moving 8 pixels simultaneously with a single **MOVQ** instruction. It also permits 16 simultaneous compares and/or merges, given the dual-pipeline MMX technology execution units and instruction "pairing". Unfortunately, 8-bit palettized rendering requires reading one byte at a time from a texture palette.

Traditionally, 8-bit rendering code for older processors played "speed tricks" which actually degrade performance on newer CPUs. These should be avoided, for high speed on the Pentium(R) processor and Dynamic Execution(TM) processors. Examples include:

- Self-modifying code, which overwrites part of the instruction stream with a new address or immediate value. This avoids the use of another register to contain data. On newer CPUs, every modified instruction can waste dozens of clocks, because deep pipelines must be emptied, and cache lines invalidated and refetched.
- Random (non-sequential) access to memory locations, as a result of doing table-lookups for multiplication, color conversions, shading, or dithering.

- Frequent data-dependent branches, especially if their success is irregular.
- Access of 32-bit registers (EAX, EBX, ECX, EDX) as 8-bit or 16-bit (AL, AH, AX, BL, etc...), to combine multiple 8-bit pixels in single 32-bit writes or to zero-out individual bytes.

System Bandwidth Considerations

3D rasterization is data-bandwidth intensive, and performance problems often occur due to inability to move data between main memory, CPU caches, CPU registers, graphics memory, and buses. As shown in Figure 4, peak bandwidths vary widely in the various pieces of the system.

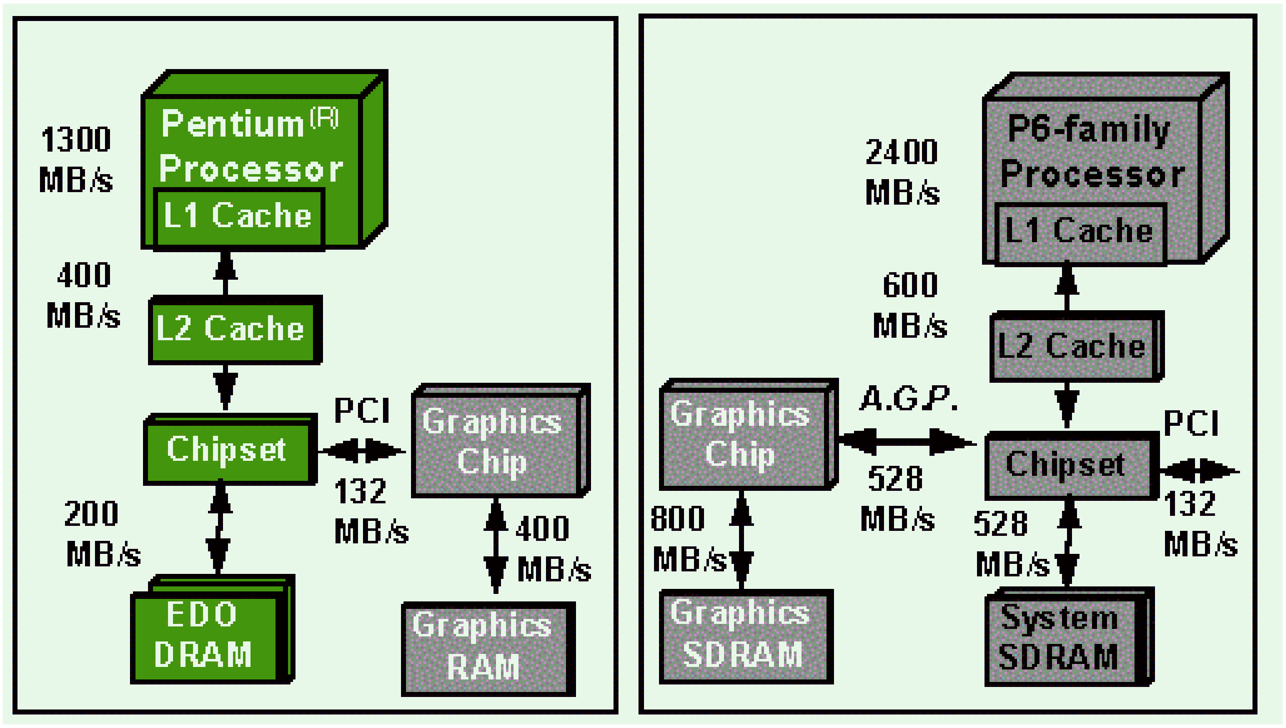


Figure 4: Peak Hardware Bandwidths for Two Typical Systems

Most PC systems have two caches, one on the CPU chip called "L1" (Level One) and a larger external cache called "L2" (Level Two). Software optimized to utilize the caches can have 2x to 10x higher performance, as the caches run 2x to 10x as fast as main memory.

Application SW cannot explicitly control the caches. It cannot command them to hold a specific data item, as the hardware attempts to keep the most useful data cache-resident.

For best performance, all accesses should be to cached data, in the order that the HW brings it into the cache. Accesses should be on the natural boundaries of the cache HW (quadwords and 32-byte cache lines). Apps should try to re-use data as much as possible once it is present in the cache. Unfortunately, SW cannot explicitly test whether particular data items are cached, and testing would cut the performance anyway. SW sequences can be cleverly crafted to get data efficiently into and out of the caches.

SW can utilize the caches efficiently by:

- Pre-fetching data several clocks before the instructions that actually use it. By putting one **MOV reg, mem** instruction in a loop, and incrementing the memory address by 32 every time, entire 32-byte cache lines will be loaded into the cache. Then subsequent uses of that data within the loop or in the next code sequence get cache hits. Of course, this technique of data "pre-touching" works well only if the algorithm can find something useful for the CPU to do during the prefetching time, which does not involve writes to memory, such as shifts or multiplies. One experiment showed a simple loop of loads and stores (using **MOVQ**) got only 58 MBytes/sec in a naive implementation. But with pre-touching, it increased to 96 MBytes/sec.

- Align large data structures on 4K Byte boundaries. Such alignment increases the likelihood of DRAM "page hits", yielding faster main memory loads into the caches.

- Access 8-bytes per instruction, or at least 4-bytes. Avoid single-byte or word moves.

- Access data sequentially in increasing and consecutive addresses. This means that when a cache line-fill occurs, the hardware probably can continue satisfying fetch-requests from the temporary line-fill buffer, rather than forcing the program to wait until the cache itself has been updated.

- Arrange data to fit within cache lines. For example, it may be beneficial to rearrange texture maps in rectangular blocks or strips or tiles, so that each cache line contains a rectangular area of the texture. If done carefully, the overhead in the texture mapping instruction loop will be small, and cache locality of texturing will improve.

- Access data on aligned boundaries. For example, **MOVQ** should always address a 0, 8, 16, 24, etc... location. If the code does an 8-byte access to a 4-byte aligned address like 28, then the access will take at least 3 extra clocks (cache gets accessed twice), and maybe more if it crosses a cache line boundary (as does an 8-byte access to address 28 crosses the line boundary at address 32.)

For more information on the CPU caches, refer to the MMX Technology Developer's Manual, chapter 3.

Other Memory Optimizations

Even for non-cached memory locations, such as graphics card RAM or uncacheable main-memory regions, SW optimization of accesses can offer speedup:

- Avoid "ping-pong" accesses to two (or more) memory arrays, either reads or writes. Most DRAM runs alot faster when "page hits" occur, meaning successive reads or writes to the same general area. So group accesses to be multiple reads to one array, followed by multiple reads to another, if two must be accessed. Ping-ponging also hampers the ability of the P6-family internal CPU write-buffers to efficiently burst stream writes to memory.
- Group writes in fours to exploit the CPU write buffers, as described in the Developers' Guide.
- Access data sequentially in increasing and consecutive addresses. (Again, to ensure DRAM page hits)
- Access data aligned to its elemental size, usually 8 bytes.
- Regroup data types in individual arrays rather than in heterogenous structures, or visa versa. The programmer should do whatever gives the most locality and sequentiality of access.

Figure 5 shows the huge variations in memory and graphics bandwidth, depending on how carefully the data are arranged and accessed. The measurements are sustained rates for very small code loops, on a Pentium(R) processor 166MHz system with:

- Windows 95
- 32 MB EDO DRAM main memory
- The 82430HX PCIset
- PBSRAM (Pipelined Burst Static RAM) 256KB L2 cache
- An S3 Virge* graphics chip with 2MB EDO graphics RAM

Systems using P6-family processors are even more sensitive to caching, alignment, and sequentiality (burst-ability) of writes.

The theoretical peak bandwidth for reads from the L1 cache would be 166 MHz * 8 bytes, or 1328 MB/s. Actual measurements were only 2/3 as large, due to the loop overhead. Also note that the uncached main memory copy numbers (at ~97 MB/s for QWORDS) were lower than expected, since they result from continual interleaving of reads and writes to different DRAM pages. To improve this bandwidth, if the source data can be cached, the algorithm may read several L1 cache lines of source from one DRAM page by touching every 32nd byte, then copying them all out of cache with sequential continuous writes. Of course, your mileage may vary.

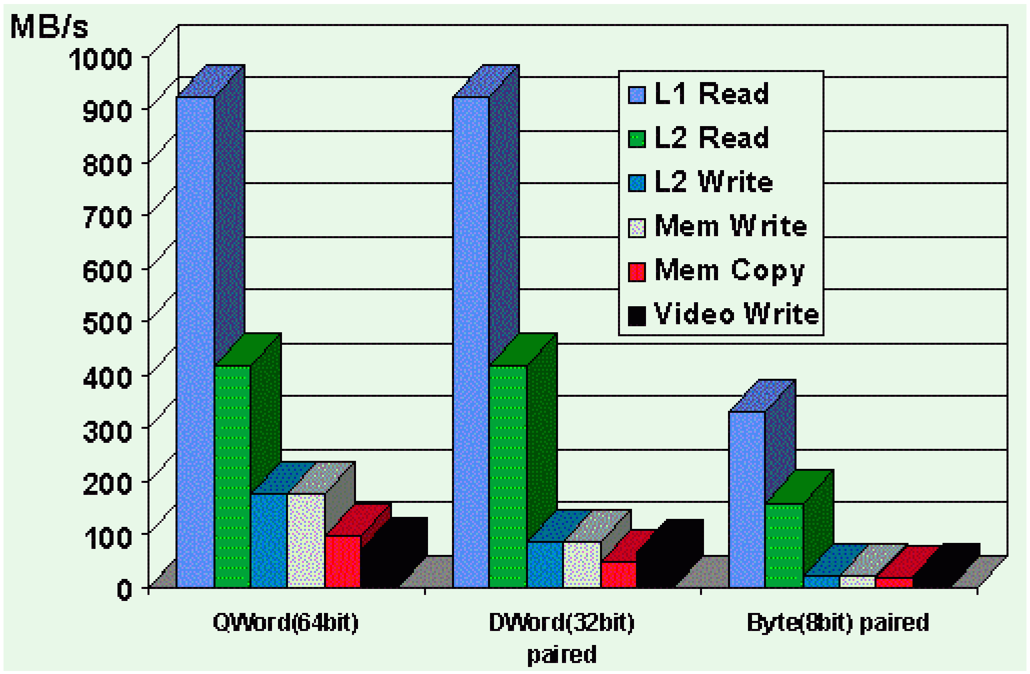


Figure 5: Measured Memory and Graphics Bandwidth for Pentium(R) Processor 166MHz 82430HX systems for various types of access

How should these maximum-bandwidth measurements be used ? Well, they tell how many CPU clocks are available for computation for each write or read. For example, writing to main memory at ~175 MBytes/sec means about one 8-byte write every 7 CPU internal clocks at 166 MHz. Thus SW can do between 7 and 15 computation instructions per write, depending on the amount of pairing. But if the computations require two inputs for every output, as in the case of alpha blending, then another 350 MBytes/sec would have to been read concurrently. If these inputs come from the L1 cache, theoretically the 175 MB/s output remains feasible. More likely, however, is reading from main memory at least one input per computation, so that the net output will drop to less than half the max measured 175 MB/s. Optimization involves balancing the computation with memory accesses, beyond just arranging the data most efficiently.

Overall Performance Predictions

Performance benefits from MMX technology in 3D rendering vary widely. Naive implementations may yield no speedup at all. Intel's VTune performance-tuning tool can help significantly in analyzing MMX technology code. Generally, speedup requires:

- Inner loops which do lots of computation for every memory access. For example, a Gouraud-shaded specular highlight fog-blended renderer should benefit. But if the same loop added Z-buffering and destination-alpha blending, it would require at least 3 more memory accesses, and probably would slowdown signficantly.
- Careful attention to data accesses, as described earlier. Avoid read/modify/write sequences which make the CPU stall waiting for memory or PCI data. Avoid unaligned accesses. Load and store data in 8-byte quantities using the **MOVQ** instruction.
- Avoiding frequent mixing of MMX instructions and floating-point instructions. Some processors impose a penalty approaching 50 clocks, to switch between the two types of operations.

Experience has shown that the **MOVQ** instruction and the "extra" registers alone can yield a 30% pixel-rate speedup in 3D rendering in real applications, whether 8-bit, 16-bit, or 24-bit coloring. For Gouraud shading at 24-bit color, MMX technology proved 2x faster than straight Intel architecture instructions. For bilinear-filtered texture mapping at 16-bit, it was more than 3x faster. For Z-buffering, overall rendering sped up from 10% to 60%, depending on the object size and the algorithm.

On two rendering libraries, we have measured on MMX technology Pentium processor 166MHz systems, complete SW pipeline pixel rates of 1 Mpix/s to 20 Mpix/s for RGB16. The higher number came from low-polygon count, non-Z-buffered, non-alpha, with a mix of flat, Gouraud, and point-sampled textured objects. The lower number came from turning on all those features and texture filtering. The numbers will improve as CPU clock rates grow, P6-family processors proliferate, and the rendering algorithms mature.

Software and Hardware Acceleration

Creating 3D applications and content for MMX technology is very much like creating for 3D hardware accelerators, as both use 16-bit or 24-bit RGB formats. In some cases, software rendering runs faster, and certainly more flexibly, than current 3D hardware. In other cases, it makes sense to combine foreground objects and special effects, rendered in software, with hardware-rendered backgrounds. Hardware imposes certain overhead in setting-up registers and synchronization, but of course has higher bandwidth to graphics memory. So MMX technology does not make 3D hardware unnecessary, nor does HW make MMX technology unnecessary.

SW 3D rasterization can accomplish some things that hardware cannot. Procedural and animated textures are one example, which create a texture mathematically in real-time. This reduces the artifacts which come from texture magnification, minification, and filtering, versus a standard bitmap-based mapper. Procedural textures can add dynamic realism, such as the appearance of wave-motion in water.

For small triangles (less than 50 pixels), the SW setup overhead for hardware accelerators may exceed the speed gain of hardware rasterization. For example, the controlling program may need to send 30-40 bytes per scanline to a low-end 3D accelerator, for each polygon. If that scanline is less than 20 pixels long, a SW-only renderer could have calculated and displayed the pixels themselves in less time. Likewise, a high-end triangle-capable 3D accelerator still needs 40-100 bytes setup per triangle, which may exceed the bytes in the rendered pixels.

Typical triangle sizes in typical scenes follow a bipolar distribution, with many small (50 pixel or less) triangles, and a few large (500-100,000 pixel) triangles. Figure 6 shows a flight simulator scene distribution: airplanes, buildings, and trees are made up of small polygons, while the ground, clouds, and sky are large polygons. In this application, SW could draw those small polys, and concurrently hardware could draw the large.

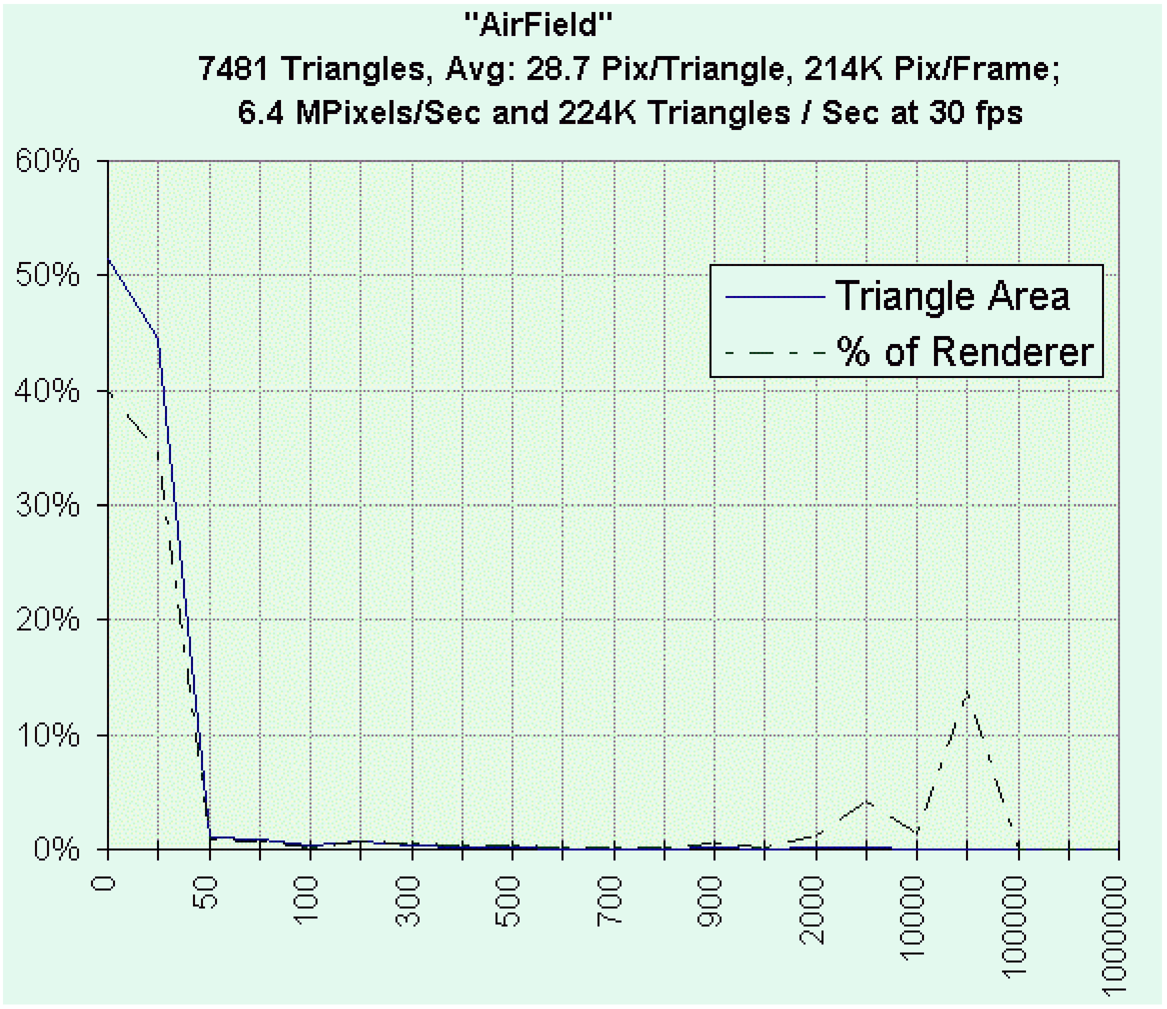


Figure 6: Polygon Size Distribution for a Sample 3D Flight Simulator Scene

Of course, the merging of SW and HW rendered polys or objects can prove complex. If they are to abut one another (E.G., SW draws a snowcap on a HW mountain), then both rasterizers must be following the same rules about subpixel locations and edge-inclusion. Furthermore, if interpenetrating objects may exist, or if SW does not pre-sort objects to ensure hidden surface removal, then a Z-buffering merge operation will be required. Finally, some locking and synchronization mechanism must be employed to prevent accelerator HW and SW (the CPU) from writing to the same pixels at the same time, and to prevent one from writing the next frame's pixels ontop of the other's current-frame pixels.

Conclusion

Adapting 3D rendering software to MMX technology can yield significant performance and quality improvements, for 16-bit and 24-bit color. To a lesser extent, even 8-bit color applications derive some speed increase.

The adaptation requires rewriting the inner loops of the 3D rasterization pipeline. Experience has shown the effort to be 4-8 person-weeks, for a typical PC game application. Some optimization of data structures may also be required -- for example, aligning arrays on 8-byte boundaries, grouping together data types in individual arrays rather than in heterogenous structures, and using 16-bit color in textures. Also, it requires tuning data access and algorithms to optimize cache utilization.

In some cases, a mix of hardware-accelerated rendering and a software pipeline can produce better results than either alone. MMX technology software rendering can implement new and/or flexible algorithms, such as procedural texturing, specular highlights, and dynamic lighting.

Appendix: Myths and Realities

Myths	**Realities**
CPU is bandwidth-bottlenecked by memory & PCI	166 MHz CPU & PCI do 640x480,30fps,16bpp (20 MPix/sec)
MMX technology is no use for colorindex (8-bit)	MMX technology can add 10%-30% for 8-bit color
Using FP (Floating Point) mixed with MMX technology is too slow (transition penalty)	In most cases, the penalty can be worked-around to rarely occur, by modularization of FP code.
MMX technology is slow for 16bit RGB565 or 555	Converting from RGB888 to 565 or 555 takes about three clks/pixel.
3D HW will make SW rendering unneeded	HW & SW are complementary. They can enhance or augment each other. SW is great at smaller objects & special effects.
MMX technology code development takes a long time.	Most apps require MMX instructions only in inner loops, and the effort is typically 4-8 person-weeks.

Appendix: MMX Technology for Geometry ?

Developers often use floating-point data and calculations for 3D Geometry, instead of integer or MMX technology. Geometry calculations include matrix & vector multiplies and additions. Reasons for floating point usage include: broader dynamic range, more precision, and simplicity.

If integer or fixed point values are used, the programmer must constrain the data to reside in a limited range, such as +/-32k for 16-bit numbers. And overflow/underflow checking must be done on products and sums.

However, for applications already written with 16-bit integer or 16-bit fixed-point geometry, MMX technology offers speedup in matrix and vector manipulations. See the app notes listed below for details:

- 3D Transform
- Efficient Vector/Matrix Multiply Routine
- Matrix Transpose demonstrates two approaches to transposing.

Appendix: Glossary of 3D Terms, and Useful Links

- Glossary of 3D Terms
- Dimension 3D Home Page for PC 3D hardware and software news and reviews
- 3D Artist Magazine Home Page
- Three D Graphics Inc, including Procedural Textures
- ACM SIGGRAPH-- the Association for Computing Machinery Special Interest Group on Graphics
- MMX Technology Manuals & General Information