Architecture of a 3D Software Stack for Peak Pentium® III Processor Performance

Paul M. Zagacki, Deep Buch,
Emile Hsieh, Daniel Melaku, Vladimir Pentkovski, Microprocessor Products Group, Intel Corp.
Hsien-Hsin Lee, EECS-ACAL, University of Michigan, Ann Arbor

Index words: 3D, Graphics, Performance, Pentium® III, Driver

ABSTRACT

In this paper, we analyze the benefits of key architectural modifications to a conventional 3D graphics software stack (application, library, and graphics driver). We do not propose a new 3D pipeline architecture; rather, we focus on improving the efficiency with which it is practically implemented. It is certainly possible to target specific layers of a 3D software stack for optimization and to realize significant performance gains with the Pentium® III processor and Internet Streaming SIMD Extensions. However, we will show that optimizing the kernel layers of the 3D software stack enables the user to take maximum advantage of the latent capabilities of the Pentium III processor. We use, as a case study, a geometry pipeline implementation, the Architecture Geometry Engine, developed by the Pentium III Architecture team (referred to as ArchGE) and a 3D scene manager. In this paper, we present performance data, based on our measurements, to demonstrate the benefit of the architectural enhancements.

INTRODUCTION

The prohibitive cost of applying the algorithms necessary to compute geometry and lighting in a conventional 3D pipeline has long kept 3D in the realm of high-end workstations. Figure 1 illustrates the classic 3D pipeline structure, which consists of several key components. First, the geometry and lighting calculations are performed on the system's host processor.[1] The application's 3D models are transformed into their virtual worlds, and lighting information is generated. These calculations are done in either a popular 3D library (OpenGL* or Microsoft's Direct3D* for example) or by the application. The generated information is then handed to another component, a 3D graphics controller, for rasterization (conversion into a 2D pixel representation of the image) on the computer screen. Keeping these two components in balance is one of the fundamental challenges that high-performance 3D engine development must address.

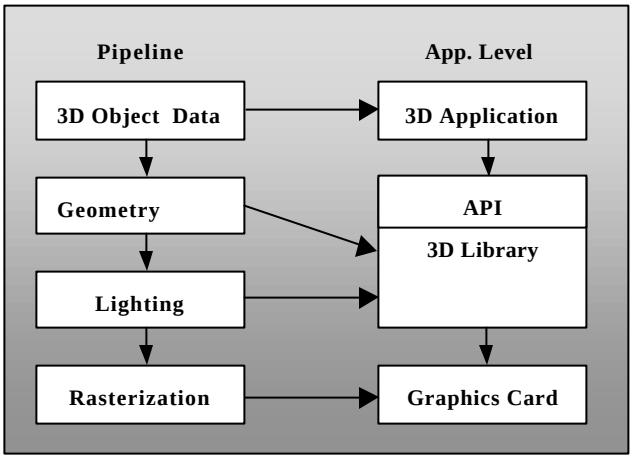


Figure 1: Typical 3D pipeline structure and its associated application-level components

An increase in graphics' controller performance means the 3D libraries built to deliver the geometry information to the cards must also increase in performance to keep the division of work in balance. The Internet Streaming SIMD Extensions were developed, in part, to increase the efficiency and throughput of the geometry and lighting calculations thus realizing higher system performance. However, to achieve peak 3D performance with the Pentium® III processor, components beyond the kernel level should also be optimized.

[1]This paper assumes a basic understanding of 3D graphics. For an in-depth review of this material refer to [1].

In order to demonstrate the peak 3D performance and usage models for the Pentium III processor, we developed the Architecture Geometry Engine (ArchGE) to fit into the 3D library layer (Figure 1). ArchGE incorporates the Internet Streaming SIMD Extensions to boost the geometry and lighting performance, but also adds two new architectural extensions to a general purpose 3D library:

1. MultiPrimitive API extension
2. "Online" driver model

The exact kernel-level speedup achieved with Internet Streaming SIMD Extensions over x87 code varies depending on the type and number of lights (infinite, local, specular, etc.), amount of clipping, primitive type, and other content variables.[2] When used in typical transformation and lighting kernels, we expect to see 1.4-2.0x the kernel-level performance over optimized x87 floating-point code for single light workloads. Workloads with multiple lights and larger primitive sizes are expected to see in excess of 2.0x the performance over optimized x87 implementations. Additionally, we have measured highly tuned, custom engines that see 2.5x–2.75x kernel-level performance. However, only a percentage of this kernel-level performance translates into application-level performance, the important measure for the consumer. The application performance increase is governed by *Amdahl's Law* and is typically less at the application level than at the kernel level unless additional optimizations to the software stack are made [2].

The MultiPrimitive API extension allows an application to gather all the primitives (e.g., strips, fans, vertex buffers) that share identical render state information and submit them in a single API call to the library. The MultiPrimitive optimization has been shown to provide 17%-40% of additional performance over conventional (single primitive per call) methods for drawing primitives. The additional performance is a result of the amortization of call overhead and vertex prefetch costs over a greater number of vertices being processed. The reduction of time spent in "startup" cost translates to more time spent in useful geometry and lighting computations that are accelerated by Streaming SIMD Extensions.

The second key extension introduced in ArchGE is an "online" driver (OLD). The OLD mechanism allows the graphics controller's driver to present the final destination buffer for the transformed and lit vertices directly to the geometry pipeline. Typically, in a general purpose API, all the vertices are transformed and lit, then placed in a buffer controlled by the library. The library signals the graphics controller when it is safe to take the buffer and render the information. When the buffer is ready, the device driver must copy the data from the library's buffer into the controller's memory (typically allocated in AGP memory). There are three issues with this methodology. First, moving large batches of transformed and lit vertices between library and graphics memory exercises the processor bus but not its computational throughput, thus leading to inefficient use of available resources. Second, this process typically generates excessive cache write-back activity (moving modified lines from a smaller, faster cache level to a larger and slower cache level or memory). This tends to aggravate the loading of the processor buses, reduce the efficiency of the cache hierarchy and prefetch instructions, and reduce the throughput of geometry computations. Third, the additional copy and formatting of the data by a typical device driver can increase driver execution times by up to 10x that of an OLD approach. This time spent in additional data movement is not time spent doing meaningful computations. OLD solves each of these issues by allowing the graphics pipeline to deposit transformed and lit vertices into the graphics controller's local memory, as they are calculated. The "direct deposit" of vertex information increases the concurrency between the geometry and lighting computations (computation intensive) with the storage of the results (bus intensive). This increased concurrency has been demonstrated to provide an additional application-level performance speedup of 30%-80% relative to a typical offline driver implementation.

The remainder of this paper discusses the following methods of 3D software stack optimizations (see Figure 1) and how these optimizations affect application-level performance:

- *3D Library/API Layer:* batch multiple primitives per drawing command, single pass vs. multiple pass geometry pipeline
- *Device Driver/Graphics Controller Layer:* online driver delivery of processed vertices
- *3D Application Layer:* object-level clipping and render state sorting

With all of these optimizations in place, ArchGE is able to display nearly 2x the peak application-level speedups of optimized x87 floating-point pipelines on similarly configured machines running identical workloads. While existing 3D libraries and device drivers are able to perform

[2]Kernel-level performance for our study is defined as including transformation, culling, specular lighting, transposition into graphics controller vertex order from a SIMD format, and storing the processed vertices to AGP memory.

the computations necessary for real-time 3D graphics, the techniques described in this paper add significantly to the overall performance for such implementations.

AMDAHL'S LAW

Amdahl's Law governs how much kernel-level speedup translates into application-level performance. Simply stated (using a 3D pipeline as an example), *Amdahl's Law* states that the amount of application-level speedup that optimizing transformation and lighting produces is limited to the percentage of time the software spends in this optimized code.

The example in Figure 2 shows an application of *Amdahl's Law* to 3D. Here we apply the Pentium® III Internet Streaming SIMD Extension instructions to transformation and lighting within a 3D application stack. We show a 2x kernel-level speedup and spend 50% of our time in these 3D geometry routines.

$$Speedup_{overall} = \frac{ExecutionTime_{old}}{ExecutionTime_{new}}$$

$$= \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

$$= \frac{1}{(1-.5) + \frac{.5}{2.0}} = 1.33x \text{ Speedup}$$

Figure 2: Amdahl's Law for predicting application-level performance applied to sample Xform and lighting optimizations

Based on Figure 2, if the performance of transform and lighting (3D library layer in Figure 1) increases 2x, yet only 50% of the total time is spent in this code, this translates to an overall application speedup of 1.33x. While a 1.33x performance increase is impressive, it is not quite the 2x we saw at the kernel level. It is clear, that in addition to porting transform and lighting routines to Internet Streaming SIMD Extensions, it may also pay off significantly to optimize other portions of the application stack to achieve peak Pentium III processor performance.

Many of the optimization techniques described in the following sections of this paper are designed to help defeat the performance-limiting affects of *Amdahl's Law* by increasing the time spent in the "enhanced" code segments.

3D LIBRARY AND API OPTIMIZATIONS

There are several popular 3D libraries and countless custom engines available to handle most of the details behind manipulating objects in three dimensions and displaying them on a 2D monitor. Existing 3D libraries typically have architectures that may potentially limit the performance an application can realize on a processor like the Pentium® III processor.

Multi-Pass Vertex Processing

Current 3D libraries normally have a multiple-pass structure for operating on input vertex information. In a multi-pass geometry pipeline, the vertices are processed through several individual loops. Each loop processes all the vertices submitted to the pipeline through transformation, backfaced culling (removal of non-forward facing triangles) and then lighting (MP half of Figure 3). There are two issues with this approach:

1. Complicated cache management code
2. Small basic code block sizes with which to interleave memory and computation instructions

Multi-pass processing is heavily dependent on cache management and potentially breaking vertex blocks, submitted for transformation and lighting, into cache sized increments. After absorbing all of the cache misses incurred during the transformation phase, the pipeline should not also have to service misses during the culling and lighting portions (even if the data stays in the L2 cache, there is still a small penalty to access it).

In addition to the extra programming efforts to directly manage cache usage for a multi-pass implementation, a small basic code block size makes it difficult to effectively interleave memory accesses and computation. Ideally, the memory leadoff/latency times for a loop should be balanced by the computation time within that loop. In a multi-pass pipe there is rarely enough computation per loop iteration to balance the load/store requirements. This makes our critical code sections memory bound and not very scalable as processor core frequency increases. (System memory performance historically lags behind processor performance.)

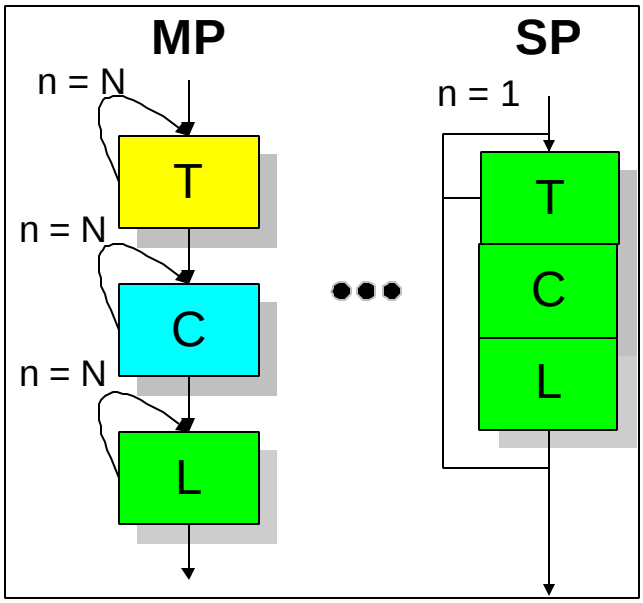


Figure 3: Multi-pass (MP) vs. single-pass (SP) geometry pipe (T = transform, C = backface cull, L = light, n= number of iterations, N = number of vertices)

Single-Pass Vertex Processing

In order to address the issues of a multi-pass pipeline structure, ArchGE implements the single-pass (SP) methodology shown in Figure 3. The key difference in this approach is that a few vertices are processed through transform, culling, lighting, and writing to the graphics controller's memory in a single loop.[3] This eliminates the need to add code to carefully manage cache utilization and also increases the basic block size significantly. The Internet Streaming SIMD Extension PREFETCH instruction is used to hide memory latency behind the computation performed in the pipe. Data, which will be transformed (x, y, z coordinate information) during the next iteration of the loop, is brought into the cache while transforming the current vertex. The same methodology applies to normals and texture coordinate(s) values during lighting computation.

By using PREFETCH instructions and implementing a large basic block, ArchGE is able to significantly increase the concurrency between the memory and computation. Our studies have shown that, as a result of this increase in coherency, a single-pass pipeline is 20%-30% faster then an optimized multi-pass pipeline. Since the ArchGE pipe tends to be more compute bound than its multi-pass counterpart, it should also scale more effectively with processor frequency.

MultiPrimitive API Extension

How vertices are submitted to a geometry pipe is almost as important as how they are processed. Most 3D libraries support many different ways to pass the application vertex information through the application programmer interface (API[4]). Vertices are grouped together into primitives (typically triangle-based) by the application and then passed to the library for transformation, lighting, and then rasterization by the graphics controller. OpenGL*, for instance, supports ten types of these primitives ranging in complexity from individual points to quadrilateral strips [4]. Since most graphics controllers accept information in triangle-based format, these are currently the most popular primitive types. Figure 4 demonstrates three such primitives.

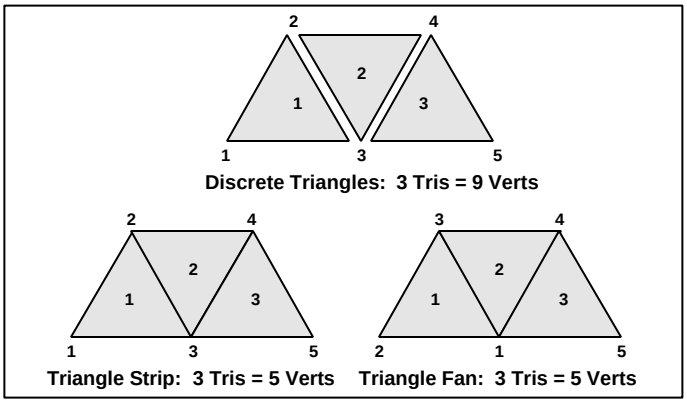


Figure 4: Different triangle primitive types and the vertices necessary to draw them

Most existing graphics libraries allow an application to submit only one primitive at a time for processing. This means that an application can only process one triangle strip, one triangle fan, or one indexed list of vertices per function call (or whatever primitives are supported by the library). Since most primitives are comprised of relatively few vertices, the overhead involved in just making the function call to process each individual primitive becomes significant.[5]

[3] In the case of ArchGE a few vertices is actually four, which nicely correlates to the Pentium® III processor's internet S.S.E. register width.

[4] The API is the set of function calls a program can make to interact with a library.

[5] This observation is based on a study of several current games and benchmarks. A similar observation was made

The overhead for processing a single primitive can be broken into two parts: additional instructions outside of geometry computations and memory de-pipelining. The obvious source of additional work is the added instructions and cycles necessary to push and pop parameters, set up transform matrices and lighting information, validate parameters, etc. This was measured to be on the order of a thousand cycles per call in some popular libraries. This is a very significant amount of time if the application is submitting a small number of vertices per call.

In the ideal case, the Pentium III processor with Internet Streaming SIMD Extensions allows for almost complete overlap of memory accesses and computation. This is achieved by fully pipelining memory accesses using the PREFETCH instruction (lower portion of Figure 5).

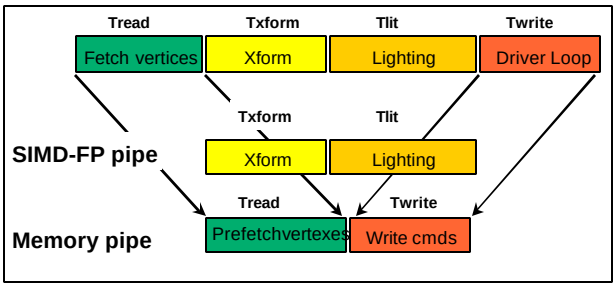


Figure 5: Ideal picture of increased memory and computational concurrency within a 3D pipeline

Each box in Figure 5 represents a block of processing time in a simplified 3D pipeline. The top portion of the figure is a conventional pipe, with serial memory and computation (a simplification since even older processor families allow for a small amount of concurrency between memory and computation). The bottom portion of Figure 5 shows what can be achieved by utilizing the PREFETCH and streaming store features of the Pentium III processor. Practically, however, an effect we refer to as "memory de-pipelining" occurs at primitive boundaries causing the total time in our ideal case to stretch somewhat [8]. For example, there can be "startup costs" associated with prefetching the first several vertices of a primitive during which computation is effectively stalled waiting for the data. For nested loops, memory de-pipelining can occur during the interval between the last iteration of an inner loop and the next iteration of its associated outer loop.

by [11] regarding some of the ViewPerf benchmark datasets.

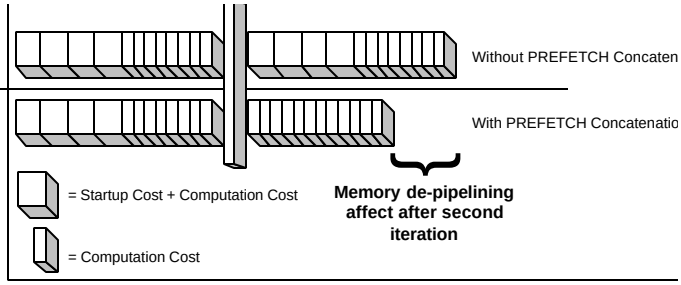


Figure 6: Memory de-pipelining between two short primitives

Figure 6 shows a graphical example of the effects of memory de-pipelining. In the figure, the large boxes represent the amount of time to do normal computation plus the time spent waiting for initial PREFETCH instructions to return data to the cache (which delays completion for several of the initial iterations of geometry processing). The smaller boxes represent the amount of time necessary to complete computation in the steady-state.

The recommended technique to alleviate the performance issue of memory de-pipelining is "prefetch concatenation." Concatenation can bridge the execution pipeline bubbles between the boundary of an inner loop and its associated outer loop by using the PREFETCH instruction to “look ahead” to the next outer loop iteration. In the example outlined in Figure 6, the geometry pipeline “looks ahead” across primitive boundaries. It is clear that if an API only allows an application to submit a single primitive per call, this technique cannot be used at primitive boundaries to amortize the memory start-up costs for each primitive submitted for processing. This is especially important when dealing with primitives containing relatively few vertices (less then 100).

In order to reduce both the impact of an application calling through the API layer for every primitive and the memory de-pipelining effects, ArchGE implements a MultiPrimitive method for passing primitives to the geometry engine. This allows an application to pass a list of primitives and a corresponding list of primitive lengths to ArchGE with one call. MultiPrimitive generates a 40% increase in application-level performance for the ArchGE/Scene Manager software stack. Figure 7 shows details of the sensitivity of MultiPrimitive to primitive size and the number of primitives in a batch. In the best case of small primitives with many primitives in each call, MultiPrimitive achieves over 400% the performance of a single primitive API. At the low end of the spectrum, very large primitives (65 – 120 vertices per primitive) with only two per call, MultiPrimitive is still able to achieve a 20% increase in application-level performance. Based on studies of existing games and benchmarks, we anticipate that this feature could potentially generate a 30%-40% application-level speedup for typical workloads.

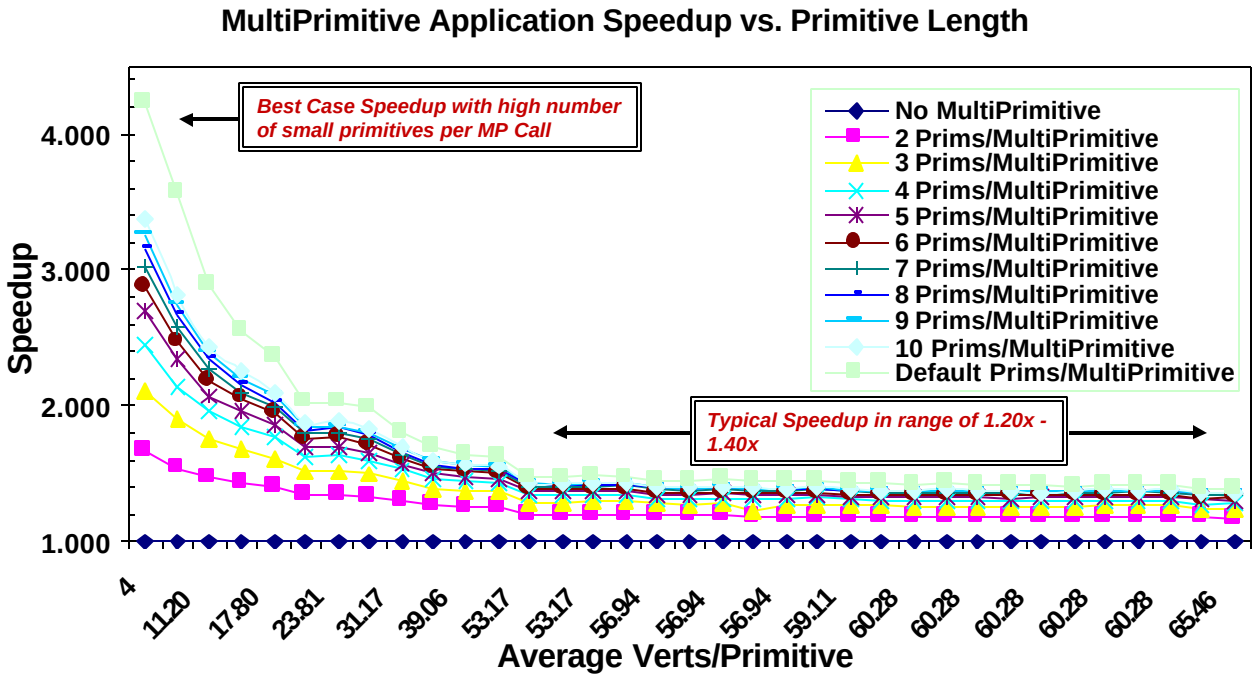


Figure 7: MultiPrimitive speedup sensitivity to primitive size and the number of primitives batched with each call (x-axis is non-linear)

The results in Figure 7 were generated by varying two variables: the maximum vertices per primitive and the number of primitives per batch. As we increase the maximum number of vertices allowed per primitive, the average number of vertices per primitive does not necessarily increase in a linear fashion (along the x-axis). The scene used for the experiment documented by Figure 7 had an original structure of 66 vertices per primitive on average. As we get closer to the original maximum average primitive size, moving right along the x-axis in Figure 7, a compression in the x-axis result occurs because additional vertices on a per-primitive basis do not affect the end average to any great extent.

DEVICE DRIVER OPTIMIZATION

Some conventional 3D libraries implement an *offline* driver model. Vertices are transformed, lit, and then stored in a temporary location within cacheable memory. The geometry engine then signals the graphics controller's driver that the buffer is ready, and the device driver begins to move the information from the temporary, cacheable memory to local memory controller memory (typically in a write-combinable and uncacheable memory range[6]). Looking back at the top portion of Figure 5, we can easily see that this will hurt the concurrency we are trying to build between memory access and computations. The conventional driver portion of the time is indicated by the "Driver Loop" time bar.

In addition to reduced concurrency within a typical geometry pipeline, *offline* driver models also have a tendency to upset the utilization of the external bus (the bus between any CPU core and memory). Many cache lines are modified in the process of storing all of the command and vertex information for a primitive (post transform and lighting information). This can easily evaporate all the careful cache utilization work done in the application and the transform and lighting routines by writing unanticipated data to the caches. Quickly, the application finds itself faced with modified cache lines that need eviction prior to pulling fresh cache lines that contain the current data necessary for computation. The modified line evictions cause an unnecessary load on internal and external busses and can significantly hurt algorithm performance.

ArchGE solved both the problem of decreased concurrency between memory and computation and the issue of inefficient cache management by implementing a different driver model. OLD differs from a conventional driver model in one simple area: a large temporary buffer for transformed and lit vertex information is not required. With OLD, vertices are transformed and lit (four at a time in our single pass implementation) and then stored immediately to memory presented by the graphics

[6] See [9] for more information on Pentium® III processor memory type definitions.

controller. ArchGE implemented an OLD mechanism for a commercially available high-performance graphics controller. In the ArchGE geometry pipeline, four vertices are transformed, lit (if visible), and deposited directly in the graphics controller's memory. Since only small blocks of vertex and command information are stored directly to memory, we have increased the concurrency between the computation in transform and lighting and have also decreased the effects of excessive cacheable writes.

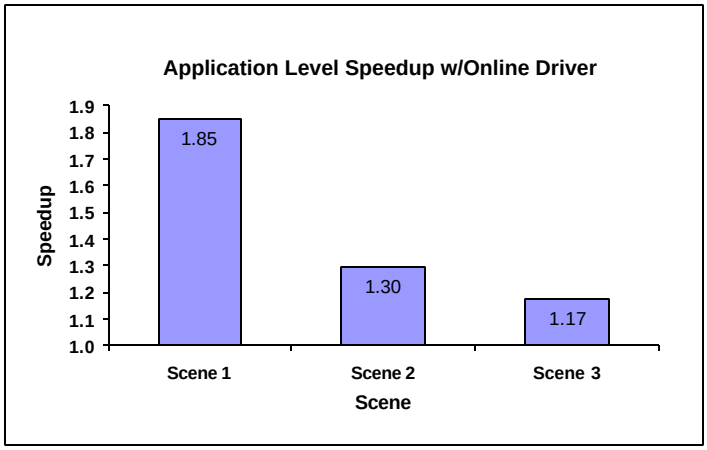


Figure 8: Application-level performance achievable with online vs. conventional driver models

Two very significant results were achieved with the implementation of an OLD in ArchGE. The first was the level of application speedup shown by this device driver model. Figure 8 demonstrates some of the possibilities of the *online* driver methodology. We measured three different scenes of varying complexity. The first scene in the chart, Scene 1, generates a 1.8x speedup over the same scene run with an offline driver in ArchGE. The high level of performance increase is attributable to the fact that the scene is very geometry intensive (approximately 82,000 vertices submitted per frame for processing) and is not bound by graphics controller fill rates (mostly small triangles). Thus, Scene 1 is more sensitive to processor capabilities and available bus bandwidth. With no external limitations on the performance of this workload, OLD is able to show close to peak performance.

The second scene in the chart, labeled Scene 2, is a close-up view of the first one and is much more sensitive to graphics controller fill rates. The somewhat lower speedup, 1.30x over a conventional driver model (vs. 1.8x for Scene 1) reflects the scene's sensitivity to fill rate. Finally, the third scene measured, Scene 3, shows a 1.17x performance delta over an offline driver. The third scene represents what we feel to be the worst-case content for ArchGE, since the geometric complexity of the content is relatively low, and the fill rate requirements are quite high, which make the graphics controller the performance bottleneck.

Figure 8 shows the large range of performance that is possible by implementing an online driver model. Our studies on the content of current games and benchmarks have indicated that results achievable fall between the peak of 1.8x and 1.3x. Tuning of the content for the third scene should yield results that fall into this range.

Increasing the amount of time spent transforming and lighting vertices is an additional effect of an online driver. With all of the additional time spent in code optimized with the Pentium® III processor's Internet Streaming SIMD Extension instructions, we are able to get much closer to the theoretical speedup generated by kernel-level optimizations of transformation and lighting (according to *Amdahl's Law* described previously). Figure 9 clearly displays the additional amount of time spent in meaningful computation in the case of the online driver. The pie-chart on the left of Figure 9 shows that 90% of our time is spent in the ArchGE library transforming and lighting vertices. In contrast, the pie-chart on the right of Figure 9 shows only 50% of our time in transformation and lighting, while we spend 46% of our time in device driver code (copying vertices to the graphics controller's local memory). Both of the profiles shown were generated using ArchGE on a very complex scene, which is not limited by graphics card fill rates.

The online driver feature translates into more time spent transforming and lighting vertices and less time moving data in and out of the cache hierarchy. The increase in focus on transformation and lighting (coupled with the optimizations possible with the Pentium III processor) allows an application to increase the level of content and generate a more realistic user experience. Our measurements have shown realistic speedups in the range of 1.8x to 1.3x with an online driver.

3D APPLICATION LAYER OPTIMIZATIONS

Optimizing a 3D application stack starts at the very top, with the application code and the content itself. The structure of the content (type of primitive, number of vertices per scene, amount of textures, etc.) has a huge impact on the performance of an application. The manner with which this content is presented to the 3D library layer is also very important. The scene manager used in our study implements a few key optimizations that generate significant benefits.

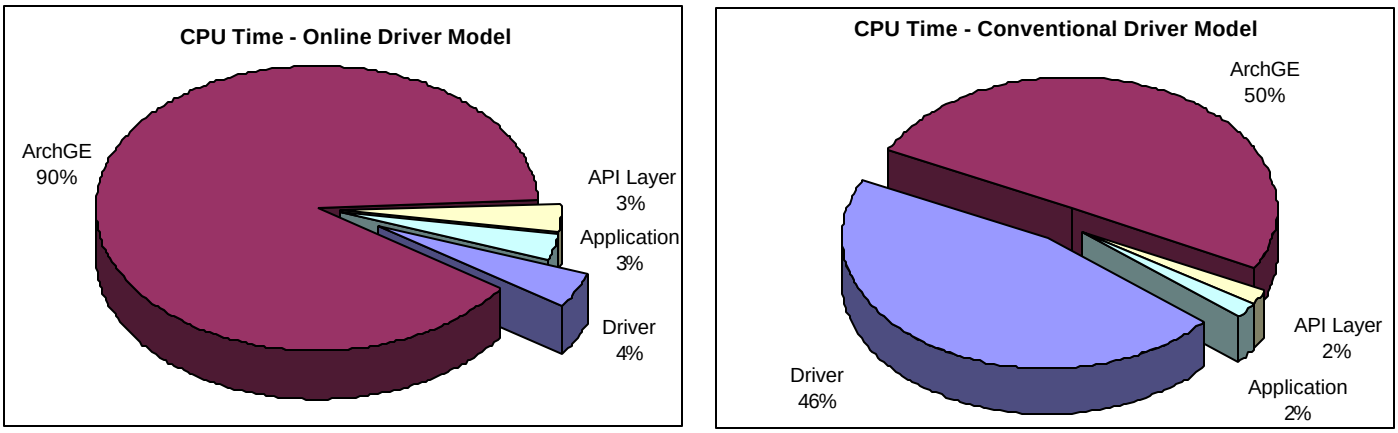


Figure 9: Effects of an online driver model to time spent in computation vs. data formating and movement

Render State Sorting

In order to convert the scene manager's 3D models into pictures on the screen, they must maintain a render state. The render state is a collection of information that tells the geometry engine and the graphics controller how to process incoming information (and display it on the screen). A render state contains information ranging from various transform matrix values to texture map addresses, which should be selected by the graphics card for rasterization.[7] Switching any portion of the render state, within a standard 3D library, is very expensive for the application.

The first cost associated with render state changes is in the 3D library itself. For the most part, a call into the 3D library to alter the current state involves a large number of processor cycles. Some of this time is spent in the library routine validating the new state and manipulating state variables. Another chunk of time goes to the device driver, where it typically goes through its own process of validation and setup.

The second cost of frequent render state changes is exhibited by the 3D graphics card. As graphics card frequencies and performance increase, so do the depths of their rasterization pipelines. It is typically necessary to flush the raster pipe of existing primitives and only restart after this has completed. This flushing leads to excessive bubbles in the rasterization pipeline and a less than effective utilization of precious pixel fill and triangle setup rate bandwidth.

By identifying the types of render states utilized and grouping primitives by distinct state setting at the time of the creation of the model/scene graph, our application was able to eliminate much of this overhead.

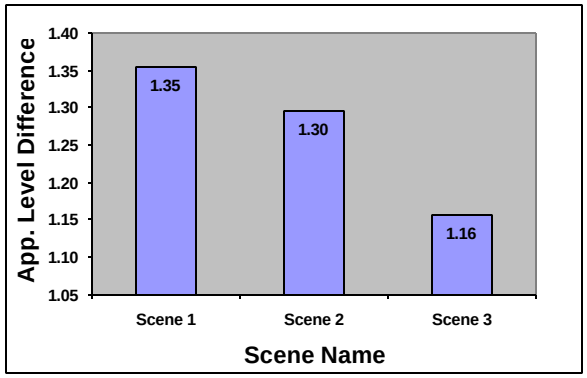
Object-Level Clipping

There are various ways a 3D application can avoid processing non-visible geometry. Our application uses bounding boxes around portions of the scene being processed to trivially accept or reject the primitives for further processing. The scene manager compares the points, which define the corners of the bounding box, with the dimensions of the viewport. There are three possible results of this comparison:

1. Completely outside of the viewing frustum; reject from further processing
2. Completely inside of the viewing frustum; submit for processing and indicate that no clipping is necessary
3. Points on the bounding box straddle the viewing frustum; submit for processing and indicate that clipping may be necessary[8]

[7] Rasterization is the process by which a graphics controller converts geometry information into pixel position and color information on a computer monitor.

[8] For more information regarding clipping primitives, please see [5] and [6].



Scene Name	Perf. Diff.	FPS w/BBoxes	FPS wout/BBoxes
Scene 1	1.35	43.6	32.19
Scene 2	1.30	47.6	36.71
Scene 3	1.16	38	32.86

Figure 10: Application-level performance difference with bounding boxes turned on

Using a bounding box with eight corner points for this comparison can save much processing time. The comparison involves transforming all the vertices for the primitive from their own object space into world or screen space. With a bounding box, only the eight corner points need to be checked. To check an individual point against an arbitrary bounding plane requires a minimum of four multiplication and three addition operations. These operations need to happen for the top, bottom, right, left, front, and back planes on the viewing frustum (a total of six planes). This becomes a total of 24 multiplication and 18 addition operations per point to account for each plane [3].

Implementing bounding boxes around 3D models minimizes the amount of computation necessary to trivially accept or reject vertices for further processing. Examples of the application-level performance impact of this optimization can be seen in Figure 10. This chart demonstrates a significant, application-level performance impact with bounding boxes enabled for object-level clipping. This effect was measured by turning the feature on or off within the application on three different scenes of varying geometric complexity. (Scene 1 contains the greatest number of vertices and Scene 3 the least.)

During our study on bounding box usage, we discovered that you can actually use too many bounding boxes around elements of a scene. The minimum number of vertices to include in a bounding box depends on many different factors and should be experimented with to determine what will work most effectively in any particular 3D software stack.

Implementing render state sorting and object-level clipping in our application layer has the potential to significantly boost the performance of the optimized ArchGE engine. The object-level clipping shows a 16%-35% application-level performance increase, and it brings ArchGE closer to peak Pentium® III processor performance by at least that much.

CONCLUSION

Software applications are exploiting more 3D graphics than ever before. The Pentium® III processor, with its Internet Streaming SIMD Extension instructions, can boost performance on 3D transformation and lighting over 2x that of optimized floating-point instructions. However, as shown in this paper, all of this kernel-level performance does not translate directly into application-level performance.

What we have outlined in this paper is a series of architectural optimizations for various levels of the 3D application software stack. Such optimizations can bring applications closer to realizing peak Pentium III performance for typical 3D graphics workloads. Utilizing a tuned scene manager and our ArchGE geometry engine, we are able to demonstrate close to 2x the application-level speedup of the Pentium® II processor at the same frequency on the Pentium III processor on a general purpose 3D software stack.

ACKNOWLEDGMENTS

All of the work and results outlined in this paper could not have been achieved without the significant help of the following people and organizations:

- **BMD Architecture:** We acknowledge Ken Castro for keeping our development and measurement process smooth with outstanding lab support.
- **MAP-PBA**: We acknowledge Shervin Kheradpir and Jeff Ma for developing the scene management software to support ArchGE and various experiments.
- **PMD Architecture**: We acknowledge Tom Huff for participating in discussions on ArchGE features, code reviews, and recommended improvements.
- **GCD:** We acknowledge Peter Doyle for his work on the definition of the i740™ definition of the online driver.
- **PDD:** We acknowledge Gerry Blank and Brandon Fliflet for prototyping and implementing the i740™ version of the online driver.

- **3dfx Interactive, Inc.**: We acknowledge Colyn Case and Andrew Hanson for helping to define and develop a Voodoo2™* version of the online driver for ArchGE that was used to generate much of the experimental data in this paper.

REFERENCES

[1] James D Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics: Principles and Practice*, Morgan Kaufmann, San Francisco, CA, pp. 29-31.

[2] David A. Patterson and John L. Hennessy, *Computer Architecture: A Quantitative Approach*, Addison-Wesley, Menlo Park, CA, pp. 201-283.

[3] David A. Patterson and John L. Hennessy, *Computer Architecture: A Quantitative Approach*, Addison-Wesley, Menlo Park, CA, pp. 868.

[4] Mason Woo, Jackie Neider, and Tom Davis, *OpenGL® Programming Guide: Second Edition*, Addison-Wesley, Menlo Park, CA, pp. 42-45.

[5] Jim F. Blinn. and Martin E. Newell, "Clipping Using Homogeneous Coordinates," *SigGraph 1978 Proceedings*, pp. 245-251.

[6] Jim F. Blinn, *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*, Morgan Kaufmann, San Francisco, CA, pp. 122-134.

[7] *Intel® Architecture Optimization Reference Manual*, available at http://developer.intel.com/design/PentiumIII/manuals/

[8] *Intel® Architecture Optimization Reference Manual*, pp. 6-13 – 6-15.

[9] *Intel® Architecture Software Developer's Manual, Volume 3: System Programming Guide*, pp. 9-4 – 9-13. Available at http://developer.intel.com/design/PentiumIII/manuals/

[10] *Intel® Architecture Optimization Reference Manual*, pp. 6-6 – 6-9.

[11] Chia-Lin Yang, Barton Sano, and Alvin R. Lebeck, "Exploiting Instruction Level Parallelism in Geometry Processing for Three Dimensional Graphics Applications," Technical Report CS-1998-14, Computer Science Department, Duke University, September 1998.

AUTHORS' BIOGRAPHIES

Paul Zagacki is a senior processor architect for the Microprocessor Products Group in Folsom, CA. He holds a B.S. degree in computer science from the University of Michigan, Ann Arbor. He has worked for Intel since 1994 in the areas of high-level performance modeling for microprocessor architectures, Pentium® III processor software and benchmark analysis and optimization, and 3D graphics implementation, performance analysis, and tuning. His professional interests include computer architecture/microarchitecture, 3D graphics, compiler performance, and software/hardware performance analysis. His e-mail is paul.zagacki@intel.com.

Deep Buch is a staff processor architect in the Microprocessor Products Group in Folsom, CA. He received an M.Tech degree in electrical engineering from the Indian Institute of Technology, Bombay, in 1989. He has been working for Intel since 1993 in the areas of processor architecture, platform technologies, and 3D graphics. Prior to joining Intel, Deep was a hardware specialist at Wipro Infotech R&D in Bangalore, India, working on ASIC and system level design. His interests are computer architecture, multimedia and communications. His e-mail is deep.k.buch@intel.com.

Emile Hsieh is a senior processor architect in the Microprocessor Product Group in Folsom, CA. He holds a B.S. degree from the National Taiwan University, Taipei, Taiwan, and a M.S. degree from Purdue University, West Lafayettte, IN, all in electrical engineering. His research interests include computer architecture, performance modeling and analysis, compilers, graphics, signal processing, and communications. His e-mail is emile.hsieh@intel.com.

Hsien-Hsin Lee is presently a Ph.D. candidate in computer science and engineering at the University of Michigan. From 1995 to 1998, Hsien-Hsin was a senior processor architect for the Microprocessor Products Group in Folsom, CA. While there he worked on design and performance modeling for the Pentium® Pro, Pentium® II and Pentium III processors. He holds a B.S.E.E. degree from the National Tsinghua University, Taiwan and an M.S.E. degree from the University of Michigan. His research interests include microarchitecture, memory system design, ILP optimization, and graphics architectures. His e-mail is linear@eecs.umich.edu.

Daniel Melaku is a processor architect for the Microprocessor Products Group in Folsom, CA. He holds a B.S. degree in computer engineering from California State University, Sacramento. Daniel has been with Intel since 1997, and has worked in the areas of performance projection, validation, and tool development. His interests

include digital signal processing, computer animation, voice and image recognition, and artificial intelligence. His e-mail is daniel.melaku@intel.com.

Vladimir Pentkovski is a Principal Engineer in the Microprocessor Product Group in Folsom. He was one of the architects in the core team that defined the Internet Streaming SIMD Extensions for the IA-32 architecture. Vladimir led the development of the Pentium III processor architecture and performance analysis. Previously he led the development of compilers and software and hardware support for programming languages for Elbrus multi-processor computers in Russia. Vladimir holds a Doctor of Science degree and a Ph.D. in computer science and engineering from Russia. His e-mail is vladimir.m.pentkovski@intel.com.