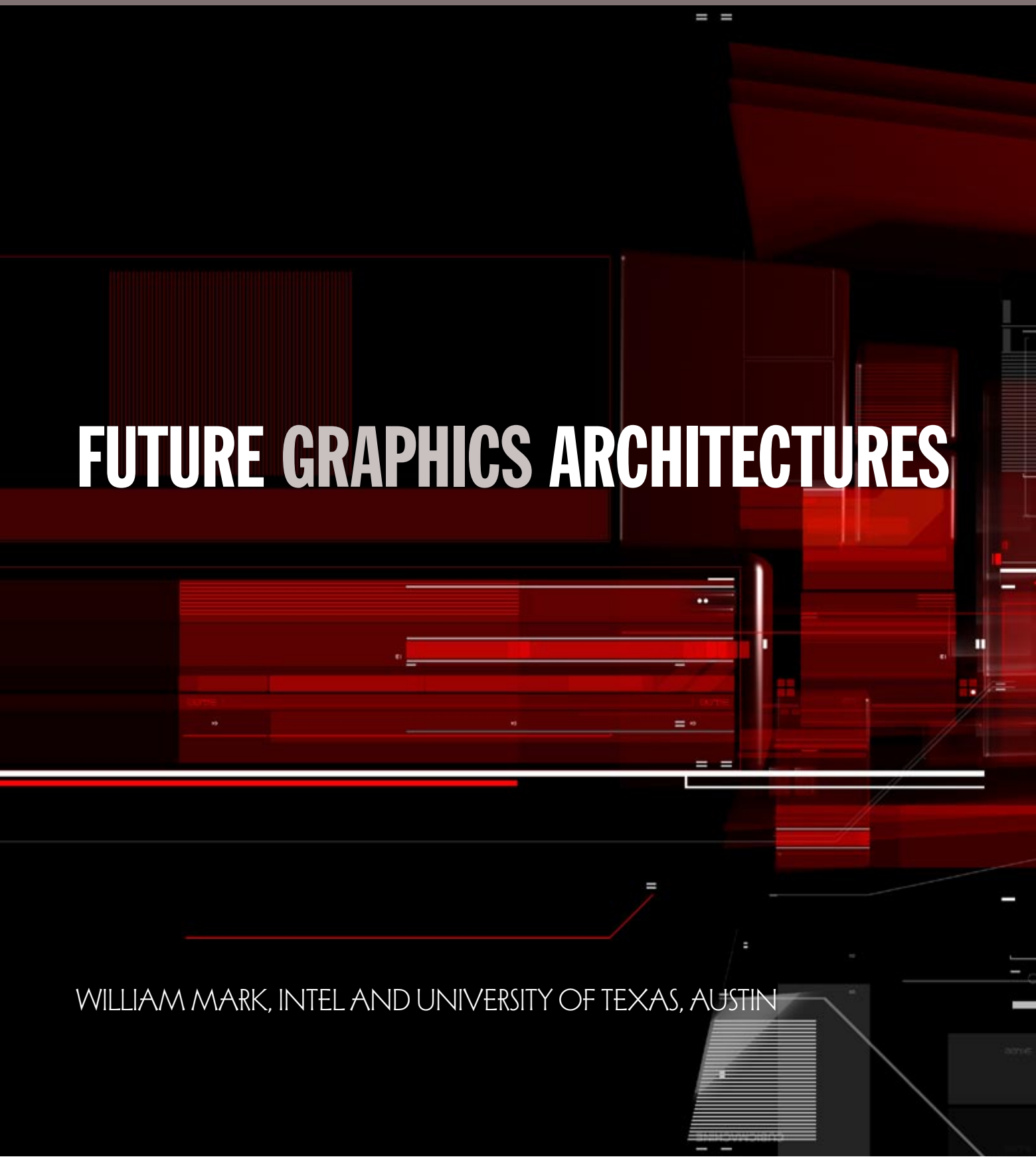
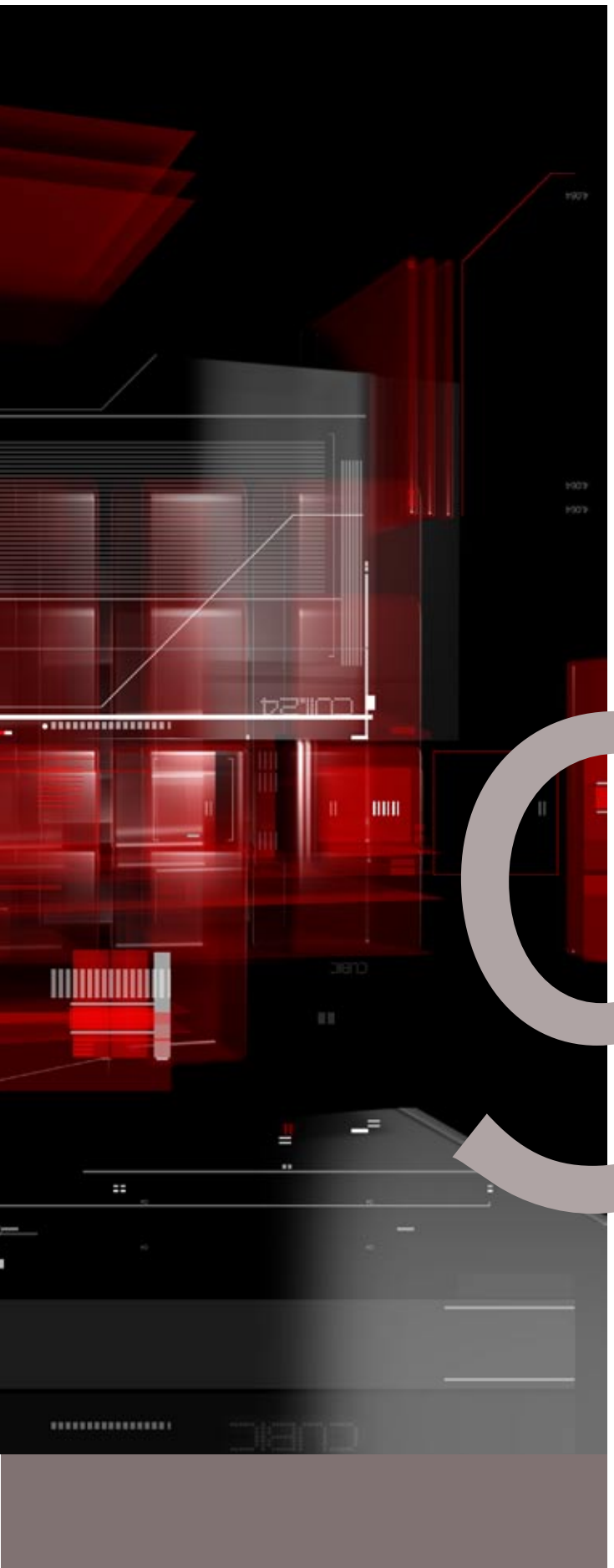


FUTURE GRAPHICS ARCHITECTURES

WILLIAM MARK, INTEL AND UNIVERSITY OF TEXAS, AUSTIN



GPUs continue to evolve rapidly, but toward what?

Graphics architectures are in the midst of a major transition. In the past, these were specialized architectures designed to support a single rendering algorithm: the standard Z buffer. Realtime 3D graphics has now advanced to the point where the Z-buffer algorithm has serious shortcomings for generating the next generation of higher-quality visual effects demanded by games and other interactive 3D applications. There is also a desire to use the high computational capability of graphics architectures to support collision detection, approximate physics simulations, scene management, and simple artificial intelligence. In response to these forces, graphics architectures are evolving toward a general-purpose parallel-programming

FUTURE GRAPHICS ARCHITECTURES

model that will support a variety of image-synthesis algorithms, as well as nongraphics tasks.

This architectural transformation presents both opportunities and challenges. For hardware designers, the primary challenge is to balance the demand for greater programmability with the need to continue delivering high performance on traditional image-synthesis algorithms. Software developers have an opportunity to escape from the constraints of hardware-dictated image-synthesis algorithms so that almost any desired algorithm can be implemented, even those that have nothing to do with graphics. With this opportunity, however, comes the challenge of writing efficient, high-performance parallel software to run on the new graphics architectures. Writing such software is substantially more difficult than writing the single-threaded software that most developers are accustomed to, and it requires that programmers address challenges such as algorithm parallelization, load balancing, synchronization, and management of data locality.

The transformation of graphics hardware from a specialized architecture to a flexible high-throughput parallel architecture will have an impact far beyond the domain of computer graphics. For a variety of technical and business reasons, graphics architectures are likely to evolve into the dominant high-throughput "manycore" architectures of the future.

This article begins by describing the high-level forces that drive the evolution of realtime graphics systems, then moves on to some of the detailed technical trends in realtime graphics algorithms that are emerging in response to these high-level forces. Finally, it considers how future graphics architectures are expected to evolve to accommodate these changes in graphics algorithms and discusses the challenges that these architectures will present for software developers.

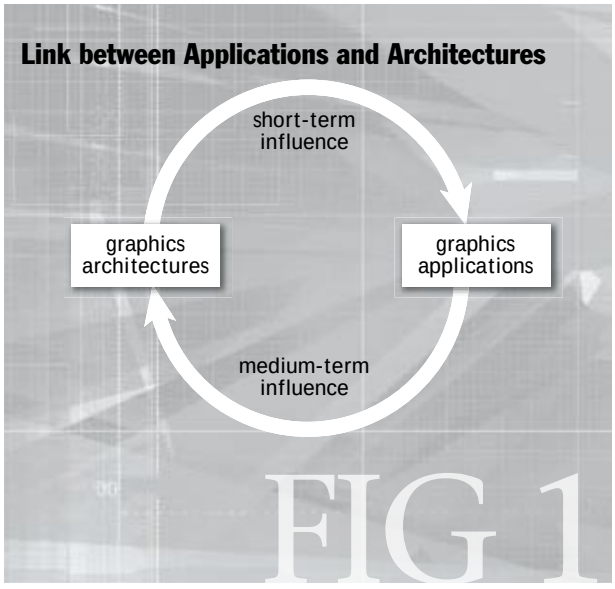
APPLICATIONS DRIVE EVOLUTION OF GRAPHICS ARCHITECTURES

To understand what form future graphics architectures are likely to take, we need to examine the forces that are driving the evolution of these architectures. As with any engineered artifact, graphics architectures are designed to deliver the maximum benefit to the end user within the fundamental technology constraints that determine what is affordable at a particular point in time. As VLSI (very large-scale integration) fabrication technology advances, the boundary of what is affordable changes, so that each generation of graphics architecture can provide additional capabilities at the same cost as the previous generation. Thus, the key high-level question is: What do we want these new capabilities to be?

Roughly speaking, graphics hardware is used for three purposes: 3D graphics, particularly entertainment applications (i.e., games); 2D desktop display, which used to be strictly 2D but now uses 3D capabilities for compositing desktops such as those found in Microsoft's Vista and Apple's Mac OS X; and video playback (i.e., decompression and display of streaming video and DVDs).

Although for most users desktop display and video playback are more important than 3D graphics, this article focuses on the needs of 3D graphics because these applications, with their significant demands for performance and functionality, have been the strongest force driving the evolution of graphics architectures.

Designing a graphics system for future 3D entertainment applications is particularly tricky because at a technical level the goals are ill defined. It is currently not possible to compute an image of the ideal quality at realtime frame rates, as evidenced by the fact that the images in computer-generated movies are of higher quality than those in computer games. Thus, designers must make approximations to the ideal computation. There are an enormous variety of possible approximations to choose



from, each of which introduces a different kind of visual artifact in the image, and each of which uses different algorithms that may in turn run best on different architectures. In essence, the system design problem is reduced to the ill-specified problem of which system (software and hardware) produces the best-quality game images for a specific cost. Figure 1 illustrates this problem. In practice, there are also other constraints, such as backward compatibility and a desire to build systems that facilitate content creation.

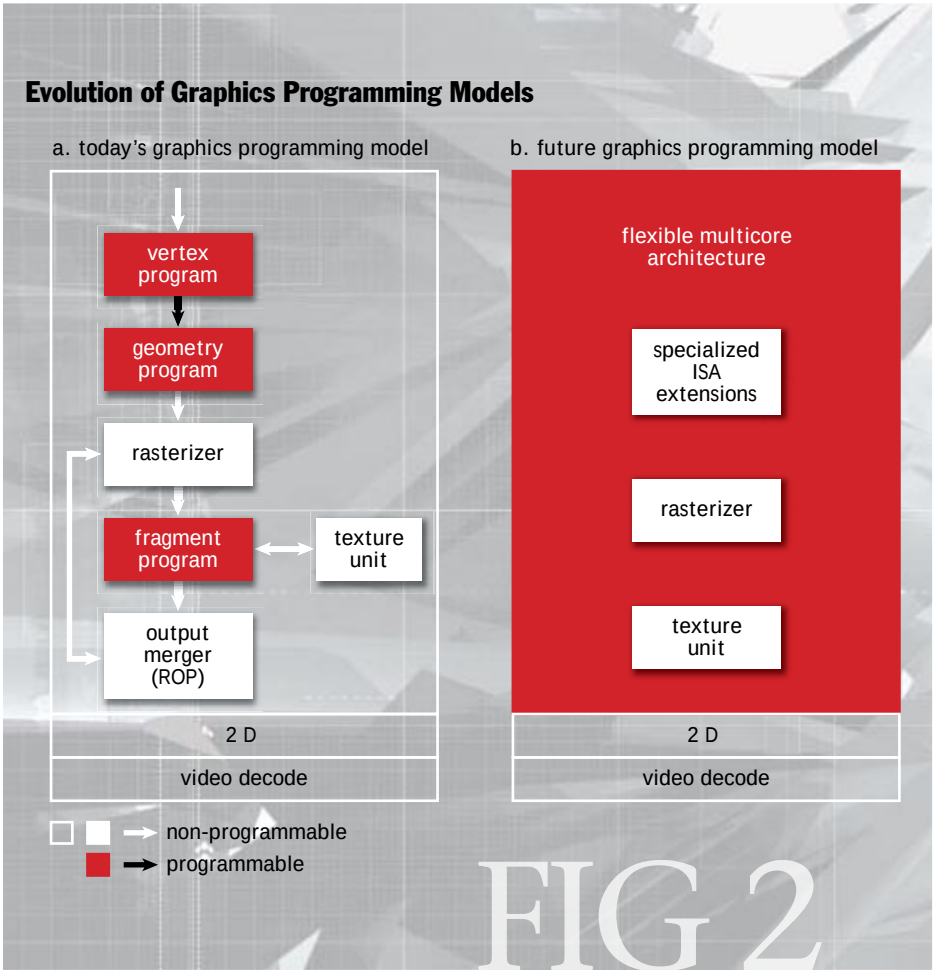
As VLSI technology advances with time, the system designer is provided with more transistors. If we assume that the frame rate is fixed at 60 Hz, the additional computational capability provided by these transistors can be used in three fundamental ways: increasing the screen resolution; increasing the scene detail (polygon count or material shader complexity); and changing the overall approximations, by changing the basic rendering algorithm or specific components of it.

Looking back at the past six years, we can see these forces at work. Games have adopted programmable shaders that allow sophisticated modeling of materials and multipass techniques that approximate shadows, reflections, and other effects. Graphics architectures have enabled these changes through the addition of programmable vertex and fragment units, as well as more flexibility in how data moves between stages in the graphics pipeline.

Current graphics processors use the programming model illustrated in figure 2a. This model supports the traditional Z-buffer algorithm and is organized around a predefined pipeline structure that is only partially reconfigurable by the application.[1] The predefined pipeline structure employs specialized hardware for the Z-buffer algorithm (in particular for polygon rasterization and Z-buffer read-modify-write operations), as well as for other operations such as the thread scheduling needed by the programmable stages.

Many of the individual pipeline stages are programmable (to support programmable material shading computations in particular), with all of the programmable stages multiplexed onto a single set of homogeneous programmable hardware processors. The programs executing within these pipeline stages, however, are heavily restricted in how they can communicate with each other and in how—if at all—they can access the global shared memory. This programming model provides high performance for the computations it is designed to support, but makes it difficult to support other computations efficiently.

It is important to realize that modern game applications fundamentally *require* programmability in the graphics hardware. This is because the real world contains an enormous variety of materials (wood, metal, glass, skin, fur, ...), and the only reasonable way to specify the



FUTURE GRAPHICS ARCHITECTURES

interactions of these materials with light is to use a different program for each material.

This situation is very different from that found for other high-performance tasks, such as video decode, which does not inherently require programmable hardware; one could design fixed-function hardware sufficient to support the standard video formats without any programmability at all. As a practical matter most video-decode hardware does include some programmable units, but this is an implementation choice, not a fundamental requirement. This need for programmability by 3D graphics applications makes graphics architectures uniquely well positioned to evolve into more general high-throughput parallel computer architectures that handle tasks beyond graphics.

LIMITS OF THE TRADITIONAL Z-BUFFER GRAPHICS PIPELINE

The Z-buffer graphics pipeline with programmable shading that is used as the basis of today's graphics architectures makes certain fundamental approximations and assumptions that impose a practical upper limit on the image quality. For example, a Z buffer cannot efficiently determine if two arbitrarily chosen points are visible from each other, as is needed for many advanced visual effects. A ray tracer, on the other hand, can efficiently make this determination. For this reason, computer-generated movies use rendering techniques such as ray-tracing algorithms and the Reyes (renders everything you ever saw) algorithm[2] that are more sophisticated than the standard Z-buffer graphics pipeline.

Over the past few years, it has become clear that the next frontier for improved visual quality in realtime 3D graphics will involve modeling lighting and complex illumination effects more realistically (but not necessarily photo-realistically) so as to produce images that are closer in quality to those of computer-generated movies. These effects include hard-edged shadows (from small lights), soft-edged shadows (from large lights), reflections from water, and approximations to more complex effects such as diffuse lighting interactions that dominate most interior environments. There is also a desire to model effects such as motion blur and to use higher-quality anti-aliasing techniques. Most of these effects are challenging to produce with the traditional Z-buffer graphics pipeline.

Modern game engines (e.g., Unreal Engine 3, CryEngine 2) have begun to support some of these effects using today's graphics hardware, but with significant limitations. For example, Unreal Engine 3 uses four different shadow algorithms, because no one algorithm provides an acceptable combination of performance and image quality in all situations. This problem is a result of limitations on the visibility queries that are supported by the traditional Z-buffer pipeline. Furthermore, it is common for different effects such as shadows and partial transparency to be mutually incompatible (e.g., partially transparent objects cast shadows as if they were fully opaque objects). This lack of algorithmic robustness and generality is a problem for both game-engine programmers and for the artists who create the game content. These limitations can also be viewed as violations of important principles of good system design such as abstraction (a capability should work for all relevant cases) and orthogonality (different capabilities should not interact in unexpected ways).

The underlying problem is that the traditional Z-buffer graphics pipeline was designed to compute visibility (i.e., the first surface hit) for regularly spaced rays originating at a single point (see figure 3a), but effects such as hard-edged shadows, soft-edged shadows, reflections, and diffuse lighting interactions all require more general visibility computations. In particular, reflections and diffuse lighting interactions require the ability to compute visible surfaces efficiently along rays with a variety of origins and directions (figure 3d). These types of visibility queries cannot be performed efficiently with the traditional graphics pipeline, but VLSI technology now provides enough transistors to support more sophisticated realtime visibility algorithms that can perform these queries efficiently. These transistors, however, must be organized into an architecture that can efficiently support the more sophisticated visibility algorithms.

Since the Z-buffer graphics pipeline is ill suited for producing the desired effects, the natural solution is to design graphics systems around more powerful visibility algorithms. Figure 3 provides an overview of some of these algorithms. I believe that these more powerful visibility algorithms will be gradually adopted over the next few years in response to the inadequacies of the standard Z buffer, although there is substantial debate in the graphics community as to how rapidly this change will occur. In particular, algorithms such as ray tracing

are likely to be adopted much more rapidly in realtime graphics than they were in movie rendering, because realtime graphics does not permit the hand-tweaking of lighting for every shot that is common in movie rendering.

THE ARGUMENT FOR GENERAL-PURPOSE GRAPHICS HARDWARE

Given the desire to support more powerful visibility algorithms, graphics architects could take several approaches. Should the new visibility techniques be implemented in some kind of specialized hardware (like today's Z-buffer visibility computations), or should they be implemented in software on a flexible parallel architecture? I believe that a flexible parallel architecture is the best choice, because it supports the following software capabilities:

Mixing visibility techniques. Flexible hardware supports multiple visibility algorithms, ranging from the traditional Z buffer to ray tracing and beam tracing. Each application can choose the best algorithm(s) for its needs. The more sophisticated of these visibility algorithms require the ability to build and traverse irregular data structures such as KD-trees, which demands a more flexible parallel programming model than that used by today's GPUs.

Application-tailored approximations. Rendering images at realtime frame rates requires making mathematical approximations (e.g., for particular lighting effects), but the variety of possible approximations is enormous. Often, different approximations use very different overall rendering algorithms and have very different performance characteristics. Since the best approximation and algorithm vary from application to application and sometimes even within an application, an architecture that allows the application to choose its approximations can provide far greater efficiency for the overall rendering task than an architecture that lacks this flexibility.

Integration of rendering with scene management. Traditionally, realtime graphics systems have used one set of data structures to represent the persistent state of the scene (e.g., object positions, velocities, and groupings) and a different set of data structures to compute visibility. The two sets of data structures are on opposite sides of an intervening API such as DirectX or OpenGL. For every frame, all of the visible geometry is transferred across this API. In a Z-buffer system this approach works because it is relatively straightforward to determine which geometry might be visible. In a ray-tracing system, however, this approach does not work very well, and it is desirable to integrate the two sets of data structures more tightly, with both residing on the graphics processor (figure 4). It is also desirable to change the traditional layering of APIs so that the game engine takes over most of the low-level rendering tasks currently handled by graphics hardware (figure 5). A highly programmable architecture makes it much easier to do this integration while still preserving flexibility for the application to maintain the persistent

Evolution of Visibility Techniques

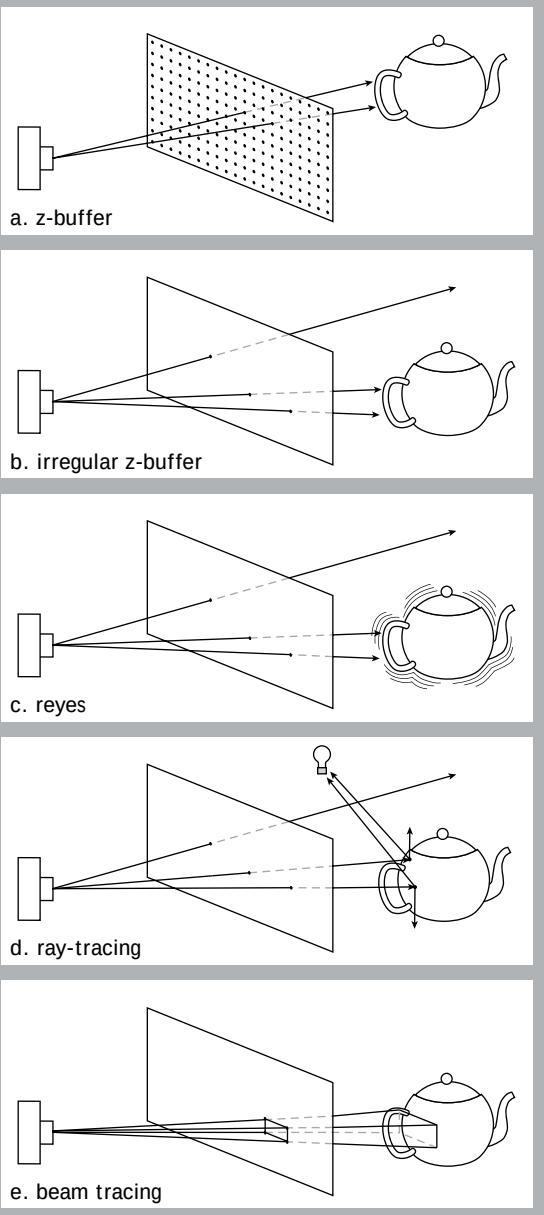


FIG 3

FUTURE GRAPHICS ARCHITECTURES

data structures in the most efficient manner. It also allows scene management computations to be performed on the high-performance graphics hardware, eliminating a bottleneck on the CPU.

Support for game physics and AI. A flexible parallel architecture can easily support computations such as collision detection, fluid dynamics simulations (e.g., for explosions), and artificial intelligence for game play. It also allows these computations to be tightly integrated with the rendering computation.

Rapid innovation. Software can be changed more rapidly than hardware, so a flexible parallel architecture that uses software to express its graphics algorithms enables more rapid innovation than traditional designs.

The best choice for the system as a whole is to use flexible parallel hardware that permits software to use aggressive algorithmic specialization and optimization, rather than to use specialized parallel hardware that mandates a particular algorithm.

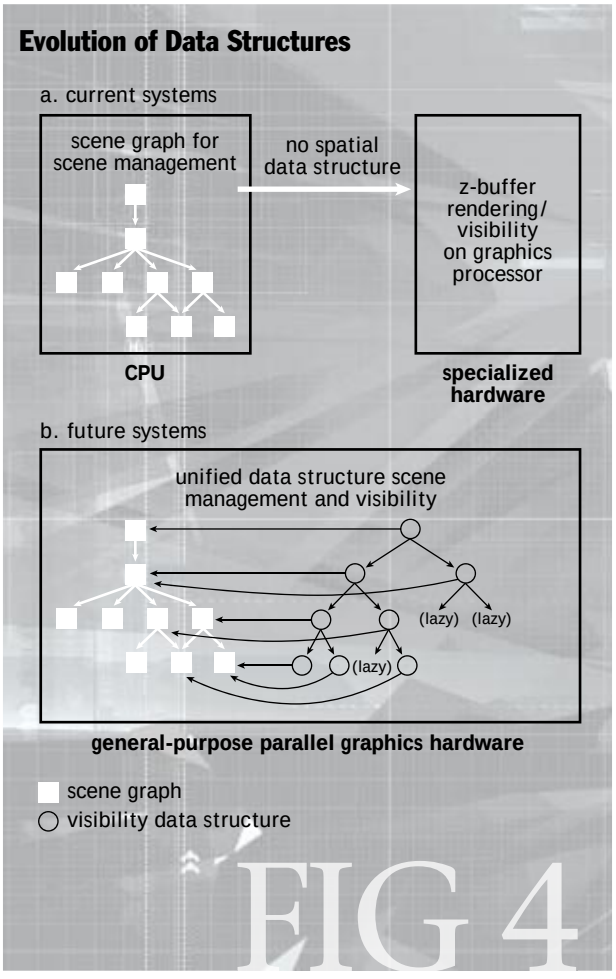


FIG 4

PROGRAMMING MODEL

When I say that future graphics architectures are likely to support an extremely flexible parallel programming model, what do I mean? There is considerable debate within the graphics hardware community as to the specific programming model that graphics architectures should adopt in the near future. I expect that in the short term each of the major graphics hardware companies will take a somewhat different path. There are a variety of reasons for this diversity: different emphasis placed on adding new capabilities versus improving performance of the old programming models; fundamental philosophical differences in tackling the parallel programming problem; and the desire by some companies to evolve existing designs incrementally.

In the longer term (five years or so), the programming models will probably converge, but there is not yet a consensus on what such a converged programming model would look like. This section presents some of the key issues that today's graphics architects face, as well as thoughts on what a converged future programming model could look like and the challenges that it

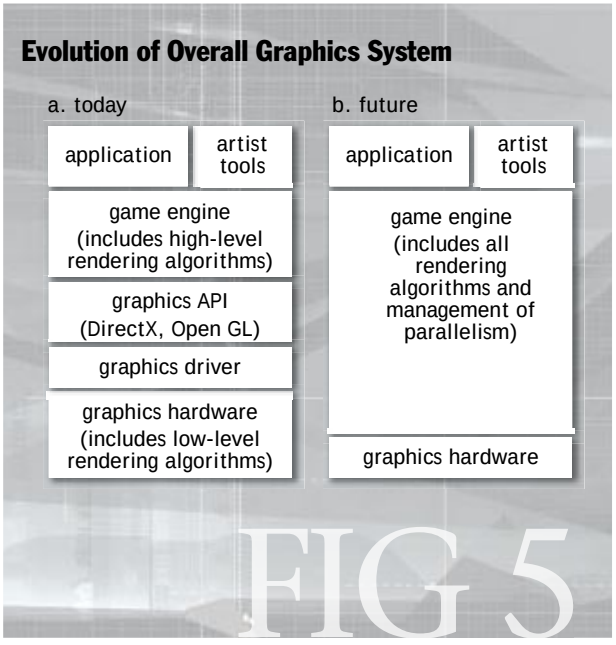


FIG 5

will present for programmers. Most of the programming challenges discussed here will be applicable to all future graphics architectures, even those that are somewhat different from the one I am expecting.

END OF THE HARDWARE-DEFINED PIPELINE

Graphics processors will evolve toward a programming model similar to that illustrated in figure 2b. User-written software specifies the overall structure of the computation, expressed in an extremely flexible parallel programming model similar to that used to program today's multicore CPUs. The user-written software may optionally use specialized hardware to accelerate specific tasks such as texture mapping. The specialized hardware may be accessed via a combination of instructions in the ISA (instruction set architecture), special memory-mapped registers, and special inter-processor messages.

The latest generation of GPUs (graphics processing units) from NVIDIA and AMD have already taken a significant step toward this future graphics programming model by supporting a separate programming model for nongraphics computations that is more flexible than the programming model used for graphics. This second programming model is an assembly-level parallel-programming model with some capabilities for fine-grained synchronization and data sharing across hardware threads. NVIDIA calls its model PTX (Parallel Thread Execution), and AMD's is known as CTM (Close to Metal). Note that NVIDIA's C-like CUDA language (see "Scalable Parallel Programming with CUDA" in this issue) is a layer on top of the assembly-level PTX. It is important to realize, however, that PTX and CTM have some significant limitations compared with traditional general-purpose parallel programming models. PTX and CTM are still fairly restrictive, especially in their memory and concurrency models.

These limitations become obvious when comparing PTX and CTM with the programming models supported by other single-chip highly parallel processors, such as Sun's Niagara server chips. I believe that the programming model of future graphics architectures will be substantially more flexible than PTX and CTM.

TASK PARALLELISM AND MULTITHREADING

The parallelism supported by current GPUs primarily takes the form of data parallelism—that is, the GPU operates simultaneously on many data elements (such as vertices or pixels or elements in an array). In contrast, task parallelism is not supported well, except for the specific case of concurrent processing of pixels and vertices. Since better support for task parallelism is necessary to support user-defined rendering pipelines efficiently, I expect that future GPUs will support task parallelism much more aggressively. In particular, multiple tasks will be able to execute asynchronously from each other and from the CPU, and will be able to communicate and synchronize with each other. These changes will require a substantially more sophisticated software runtime environment than the one used for today's GPUs and will introduce significant complexity into the hardware/software interactions for thread management.

As with today's GPUs and Sun's Niagara processor, each core will use hardware multithreading,[3] possibly augmented by additional software multithreading along the lines of that used by programmers of the Cell architecture. This multithreading serves two purposes:

- First, it allows the core to remain fully utilized even if each individual instruction has a pipeline latency of several cycles—the core just executes an instruction from another thread.
- Second, it allows the core to remain fully utilized even if one or more of the threads on the core stalls because of an off-chip DRAM access such as those that occur when fetching data from a texture. Programmers will face the challenge of exposing parallelism for multiple cores *and* for multiple threads on each core. This challenge is already starting to appear with programming models such as NVIDIA's CUDA.

SIMD EXECUTION WITHIN EACH CORE

An important concern in the design of graphics hardware is obtaining the maximum possible performance using a fixed number of transistors on a chip. If one instruction cache/fetch/decode unit can be shared among several arithmetic units, the die area and power requirements of the hardware are reduced, as compared with a design that has one instruction unit per arithmetic unit. That

FUTURE GRAPHICS ARCHITECTURES

is, a SIMD (single instruction, multiple data) execution model increases efficiency as long as most of the elements in the SIMD vectors are kept active most of the time. A SIMD execution model also provides a simple form of fine-grained synchronization that helps to ensure that memory accesses have good locality.

Current graphics hardware uses a SIMD execution model, although it is sometimes hidden from the programmer behind a scalar programming interface as in NVIDIA's hardware. One area of ongoing debate and change is likely to be in the underlying hardware SIMD width; there is a tension between the efficiency gained for regular computations as SIMD width increases and the efficiency gained for irregular computations as SIMD width decreases. NVIDIA GPUs (GeForce 8000 and 9000 series) have an effective SIMD width of 32, but the trend has been for the SIMD width of GPUs to decrease to improve the efficiency of algorithms with irregular control flow.

There is also debate about how to expose the SIMD execution model. It can be directly exposed to the programmer with register-SIMD instructions, as is done with x86 SSE instructions, or it may be nominally hidden from the programmer behind a scalar programming model, as is the case with NVIDIA's GeForce 9000 series. If the SIMD execution model is hidden, the conversion from the scalar programming model to the SIMD hardware may be performed by either the hardware (as in the GeForce 9000 series) or a compiler or some combination of the two. Regardless of which strategy is used, programmers who are concerned with performance will need to be aware of the underlying SIMD execution model and width.

SMALL AMOUNTS OF LOCAL STORAGE

One of the most important differences between GPUs and CPUs is that GPUs devote a greater fraction of their transistors to arithmetic units, whereas CPUs devote a greater fraction of their transistors to cache. This difference is one of the primary reasons that the peak performance of a GPU is much higher than that of a CPU.

I expect that this difference will continue in the future. The impact on programmers will be significant: although the overall programming model of future GPUs will become much closer to that of today's CPUs, programmers will need to manage data locality much more carefully on future GPUs than they do on today's CPUs.

This problem is made even more challenging by multithreading; if there are N threads on each core, the amount of local storage per thread per core is effectively 1/N of the core's total local storage. This issue can be mitigated if the N threads on a core are sharing a working set, but to do this the programmer must think of the N threads as being closely coupled to each other. Similarly, programmers will have to think about how to share a working set across threads on different cores.

These considerations are already becoming apparent with CUDA. The constraints are likely to be frustrating to programmers who are accustomed to the large caches of CPUs, but they need to realize that extra local storage would come at the cost of fewer ALUs (arithmetic logic units), and they will need to work closely with hardware designers to determine the optimum balance between cache and ALUs.

CACHE-COHERENT SHARED MEMORY

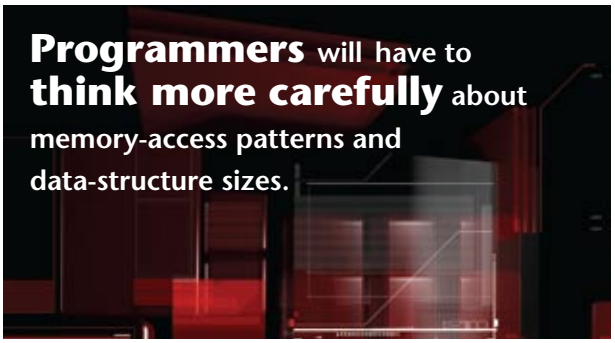
The most important aspect of any parallel architecture is its overall memory and communication model. To illustrate the importance of this aspect of the design, consider four (of many) possible alternatives (of course, hybrids and enhancements of these models are possible):

- A message-passing architecture, in which each processor core has its own memory space and all communication occurs through explicit message passing. Most large-scale supercomputers (those with 100-plus processors) use this model.
- An architecture such as the Sony/Toshiba/IBM Cell with a noncached, noncoherent shared memory. In such an architecture, all transfers of data between a core's small private memory and the global memory must be orchestrated through explicit memory-transfer commands.
- An architecture such as NVIDIA's GeForce 8800 with what amounts to a minimally cached, noncoherent shared memory, with support for load/store to this memory.
- An architecture such as modern multicore CPUs, with cached, coherent shared memory. In such architectures, hardware mechanisms manage transfer of data between cache and main memory and ensure that data in caches of different processors remains consistent.

There is considerable debate within the graphics architecture community as to which memory and communication model would be best for future architectures, and

in the near term different hardware vendors are taking different approaches. Software programmers should think carefully about these issues so that they are prepared to influence the debate.

Which approach is most likely to dominate in the medium to long term? I have previously argued that the trend in rendering algorithms is toward those that build and traverse irregular data structures. These irregular data structures allow algorithms to adapt to the scene geometry and the current viewpoint. Explicitly managing all data locality for these algorithms is painful, especially if multiple cores share a read/write data structure. In my experience, it is easier to develop these algorithms on a cache-coherent architecture, even if achieving optimal



performance often still requires thinking very carefully about the communication and memory-access patterns of the performance-critical kernels.

For these and other reasons too detailed to discuss here, I believe that future graphics architectures will efficiently support a cache-coherent memory model, and that any architecture lacking these capabilities will be a second choice at best for programmers who are developing innovative rendering techniques. Sun's Niagara architecture provides a good preview of the kind of memory and threading model that I anticipate for future GPUs. I also expect, however, that cache-coherent graphics architectures will include a variety of mechanisms that provide the programmer with explicit control over communication and memory access, such as streaming loads that bypass the cache.

FINE-GRAINED SPECIALIZATION

The desire to support greater algorithmic diversity will drive future graphics architectures toward greater flexibility and generality, but specialization will still be used where it provides a sufficiently large benefit for the majority of applications. Most of this specialization will be at a fine granularity, used to accelerate specific operations, in contrast to the coarse, monolithic granularity used to dictate the overall structure of the algorithms executed on the hardware in the past.

In particular, I expect the following specialization will continue to exist for graphics architectures:

Texture hardware. Texture addressing and filtering operations use low-precision (typically 16-bit) values that are decompressed on the fly from a compressed representation stored in memory. The amount of data accessed is large and requires multithreading to deal effectively with cache misses. These operations are a significant fraction of the overall rendering cost and benefit enormously from specialized hardware.

Specialized floating-point operations. Rendering makes heavy use of floating-point square-root and reciprocal operations. Current graphics hardware provides high-performance instructions for these operations, as well as other operations used for shading such as swizzling and trigonometric functions. Future graphics hardware will need to do the same.

Video playback and desktop compositing. Video playback and 2D and 2.5D desktop window operations benefit significantly from specialized hardware. Specialization of these operations is especially important for power efficiency. I anticipate that much of this hardware will follow the traditional coarse-grained monolithic fixed-function model and thus will not be useful for user-written 3D graphics programs.

Current graphics hardware also includes specialized hardware to assist with triangle rasterization, but I expect that this task will be taken over by software within a few years. The reason is that rasterization is gradually becoming a smaller fraction of total rendering costs, so the penalty for implementing it in software is decreasing. This trend will accelerate as more sophisticated visibility algorithms supplement or replace the Z buffer.

As graphics software switches to more powerful visibility algorithms such as ray tracing, it may become clear that certain operations represent a sufficiently large portion of the total computation cost that hardware acceleration would be justified. For example, future architectures could include specialized instructions to accelerate the data-structure traversal operations used by ray tracing.

THE CHALLENGE FOR GRAPHICS ARCHITECTS

At a high level, the key challenge facing future graphics architectures is to strike the best balance between the desire to provide high performance on existing graphics algorithms and the desire to provide the flexibility needed to support new algorithms with high perfor-

FUTURE GRAPHICS ARCHITECTURES

mance, including nongraphics algorithms and the next generation of more capable and sophisticated graphics algorithms. I believe that the opportunity for improved visual quality and robustness provided by more sophisticated graphics algorithms will cause the transition to more flexible architectures to happen relatively rapidly, an opinion that remains a matter of debate within the graphics architecture community.

THE FUTURE OF GRAPHICS ARCHITECTURES

In the past, graphics architectures defined the algorithms used for rendering and their performance. In the future, graphics architectures will cease to define the rendering algorithms and will simply set the performance and power efficiency limits within which software developers may do whatever they want.

For the programmer, future graphics architectures are likely to be very similar to today's multicore CPU architectures, but with greater SIMD instruction widths and the availability of specialized instructions and processing units for some operations. Like today's Niagara processor, however, the amount of cache per processor core will be relatively small. To achieve peak performance, programmers will have to think more carefully about memory-access patterns and data-structure sizes than they have been accustomed to with the large caches of modern CPUs.

Future graphics architectures will enable a golden age of innovation in graphics; I expect that over the next few years we will see the development of a variety of new rendering algorithms that are more efficient and more capable than the ones used in the past. For computer games, these architectures will allow game logic, physics simulation, and AI to be more tightly integrated with rendering than before. For data-visualization applications, these architectures will allow tight integration of domain-specific data analysis with the rendering computations used to display the results of this analysis. The general-purpose nature of these architectures combined with the low cost enabled by their high-volume market will also cause them to become the preferred platform for almost all high-performance floating-point computations. Q

ACKNOWLEDGMENTS AND FURTHER READING

Don Fussell, Kurt Akeley, Matt Pharr, Pat Hanrahan, Mark Horowitz, Stephen Junkins, and several graphics hardware architects contributed directly and indirectly to the ideas in this article through many fun and productive discussions. More details about many of the ideas discussed in this article can be found in another article I wrote with Don Fussell in 2005.[4] The tendency of graphics hardware to become increasingly general until the temptation emerges to incorporate new specialized units has existed for a long time and was described in 1968 as the "wheel of reincarnation" by Myer and Sutherland.[5] The fundamental need for programmability in realtime graphics hardware, however, is much more important now than it was then.

REFERENCES

1. Blythe, D. 2006. The Direct3D 10 system. In *ACM SIGGRAPH 2006 Papers*: 724–734.
2. Cook, R.L., Carpenter, L., Catmull, E. 1987. The Reyes image rendering architecture. *Computer Graphics* (*Proceedings of ACM SIGGRAPH*): 95–102.
3. Laudon, J., Gupta, A., Horowitz, M. 1994. Interleaving: a multithreading technique targeting multiprocessors and workstations. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*: 308–318.
4. Mark, W., Fussell, D. 2005. Real-time rendering systems in 2010. Technical Report 05-18, University of Texas.
5. Myer, T.H., Sutherland, I.E. 1968. On the design of display processors. *Communications of the ACM*, 11(6): 410–414.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

BILL MARK leads Intel's advanced graphics research lab. He is on leave from the University of Texas at Austin, where until January 2008 he led a research group that investigated future graphics algorithms and architectures. In 2001-2002 he was the technical leader of the team at NVIDIA that co-designed (with Microsoft) the Cg language for programmable graphics hardware and developed the first release of the NVIDIA Cg compiler. His research interests focus on systems and hardware architectures for realtime computer graphics and on the opportunity to extend these systems to support more general parallel computation and a broader range of graphics algorithms, including interactive ray tracing.