University of Texas at Austin Dept. of Computer Sciences Technical Report #TR-05-18, May 2, 2005

Real-Time Rendering Systems in 2010

William R. Mark [*] Donald Fussell [†]

Department of Computer Sciences
The University of Texas at Austin

Abstract

We present a case for future real-time rendering systems that support non-physically-correct global illumination techniques by using ray tracing visibility algorithms, by integrating scene management with rendering, and by executing on general-purpose single-chip parallel hardware (CMP's). We explain why this system design is desireable and why it is feasible. We also discuss some of the research questions that must be addressed before such a system can become practical.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture— [I.3.2]: Computer Graphics—Graphics Systems

1 Introduction

For many years, real-time graphics systems have used the traditional Z-buffer pipeline model, which is limited to local illumination computations. With appropriate modifications, this pipeline can support some restrictive global illumination techniques, but doing so is awkward and often inefficient. A different strategy is possible – VLSI technology has now progressed to the point where we are on the verge of having sufficient raw computational capability to use more general global illumination techniques. But there is no consensus yet about how future graphics systems supporting global illumination should be organized.

If we look a few years into the future, several major questions become evident: What rendering algorithms are most appropriate for this new era? What architectures should we build to support these algorithms? And what overall system organization should tie together the application, rendering algorithms, and hardware? We believe that these questions have not yet been answered satisfactorily.

The purpose of this paper is to argue that these questions are closely coupled and that addressing them will require simultaneous investigation of software algorithms and hardware architectures. We also propose a set of algorithmic and architectural approaches that we believe present one promising avenue of investigation. Our hope is that this paper will stimulate discussion in the research community and help to inspire the combined software and hardware research that we believe is critical to forward progress.

The application-level goal that drives our investigation is support for real-time global illumination for dynamic scenes. We place greater emphasis on non-physically-correct global illumination techniques than on fully physically-based techniques, since non-physically-correct techniques represent an intermediate step between today's local illumination models and eventual use of 100% physically-based techniques.

Most global illumination techniques require a more general visibility-computation capability than that provided by today's Z buffer. We present an algorithmic approach organized around ray tracing visibility algorithms that efficiently supports dynamic scenes by integrating scene management with rendering. But this tighter integration requires that the graphics hardware directly support model management as well as rendering.

At the hardware level, we advocate a very flexible architecture: a multi-core, multi-threaded, MIMD architecture with coherent access to a single address space. This architecture efficiently supports application-specific scene management code as well as the creation and traversal of dynamic, irregular data structures.

1.1 Background

The Z-buffer 3D graphics pipeline has been widely used for more than 20 years. As VLSI technology has advanced, this system organization has progressed down the cost curve from multimillion dollar flight simulators, through high-end graphics workstations made by companies such as SGI (e.g. [Akeley 1993]), down to single-chip GPUs made by companies such as NVIDIA and ATI.

For most of this history, Z-buffer graphics hardware was configurable but not programmable. However, over the past four years, we have seen the introduction of user-accessible programmability at both the vertex [Lindholm et al. 2001] and fragment [NVIDIA Corp. 2003] stages of the pipeline. The vertex programmability merely exposed a programmable engine that had already existed in various forms for many years, but the fragment programmability exposed fundamentally new hardware functionality. Its introduction was driven by the realization that beyond a certain point, the best way to use additional VLSI transistors to improve image quality is to increase the quality of each pixel rather than increasing the number of pixels or increasing the geometric detail.

Fragment programmability enabled commodity real-time systems [Mark et al. 2003] to support programmable shading capabilities inspired by those of Renderman [Hanrahan and Lawson 1990]. However, this programmability has proven to be sufficiently flexible that researchers have begun to think of graphics processors as general-purpose stream processors [Kapasi et al. 2002], capable of supporting a variety of non-shading computations [Purcell et al. 2002; Thompson et al. 2002; Bolz et al. 2003]. But at the current time, most of these other uses of the GPU are not yet fully practical. The reason is that the current GPU programming model has limitations that limit performance on general-purpose computations to much less than peak performance. We expect that this situation will change with time, but not as rapidly as many researchers are expecting.

Thus, the primary economic force driving GPU design is still real-time rendering, which leads us to the following question: What rendering requirements should drive the future evolution of graphics hardware? Another way of asking this question is, what additional capabilities could best be put to good use by applications? Of course, it only makes sense to consider capabilities that have the potential to be cost effective in the time frame of interest.

2 Application needs

We believe that there is still unmet application demand for higher-fidelity real-time imagery. For example, most observers would

[*]e-mail: billmark@cs.utexas.edu
[†]e-mail: fussell@cs.utexas.edu

agree that the images produced by batch-rendering systems are noticeably superior to those produced by interactive graphics systems, and that they would like to see these higher-quality images produced by real-time systems.

Some of the demand for improved image quality in real-time graphics can be met by adding support for object-space shading like that used in batch rendering systems such as REYES [Cook et al. 1987]. In particular, REYES provides better temporal and spatial anti-alising than the screen-space shading used in current real-time graphics systems. However, much of the current difference between batch rendering and real-time rendering results from the poor modeling of global illumination effects in real-time rendering systems as compared to batch rendering systems. We are already seeing demand for realistic global illumination with the current focus on the special case of real-time hard shadow generation. REYES and similar systems do not support global illumination computations in any general sense.

Some observers argue that REYES and similar algorithms can be used to fake a wide variety global illumination effects, as demonstrated by their use for over 10 years for movie rendering. However, interactive graphics applications are fundamentally different from batch movie rendering because the viewpoints and scene configurations are not known a-priori by the programmers and artists. Most of the techniques used to fake global illumination with REYES-like systems rely on viewpoint-dependent hand tuning and thus are not appropriate for use in real-time graphics.

2.1 Use ray tracing visibility

Almost all algorithms that model global illumination effects without the use of extensive hand-tuning rely on global visibility computations. Examples include radiosity, ray tracing [Whitted 1980], photon mapping [Wann Jensen 2001], approximation of far-field illumination using spherical basis functions, etc. Thus, we believe that robust support for global illumination requires support for global visibility computations, and specifically for ray tracing visibility.

Recent work shows that raw computational capability has now advanced to the point where it is reasonable to consider using ray tracing visibility in real-time graphics systems. Over the past several years, several groups have built near-real-time ray tracing systems with steadily improving price/performance ratios. Most of these systems run on standard CPUs (e.g. [Parker et al. 1999a; Parker et al. 1999b; Hurley et al. 2002; Wald et al. 2003b], but one runs on a specialized ray tracing architecture implemented with an FPGA [Schmittler et al. 2004], and another uses the fragment processors of mainstream GPUs [Purcell et al. 2002]. The system with the best price/performance ratio [Hurley 2005] runs on a desktop PC with frame rates over 30 frames/sec for eye+shadow rays on complex scenes. Its raw performance has been quoted at over 100M Ray segments/sec. A recent review article [Wald et al. 2003c] provides an excellent overview of recent developments in this area.

2.2 Use non-physically-correct global illumination

Experience has proven [Gritz and Hahn 1996; Kato 2002] that ray tracing algorithms and variants such as photon mapping provide the most robust and general solution to the global illumination problem. However, we do not expect 2010-era real-time game applications to rely primarily on *physically correct* global illumination. Instead, we expect that these applications will use the point-to-point visibility queries enabled by a ray tracing visibility framework to implement various non-physically correct approximations to global illumination. For example, we expect techniques such as ambient occlusion [Moyer 2004], instant radiosity [Keller 1997], and variations of precomputed radiance transfer [Sloan et al. 2002] to be used. For most of its history, computer graphics has relied heavily on phenomenological or quasi-physical approximations to illumination computation, and we do not expect that situation to change immediately. In fact, we expect that new phenomenological approximation techniques will be developed that leverage the capabilities of a ray tracing visibility engine.

2.3 Dynamic scenes are the challenge

Most interactive applications, particularly those in the economically important gaming market, use dynamic scenes. These scenes include geometrically complex objects that move, and, more significantly, deform. Unfortunately, there has been very little effort devoted to raytracing for dynamic scenes, and in particular for deformable objects.

Deformable objects such as the skinned characters [Lander 1998] used in QuakeIII and Doom present a significant challenge. The deformable nature of these characters is not well supported by any existing method for ray tracing. In particular, the simple approach of pre-building an acceleration data structure for the object and repositioning that object within the scene [Lext and Akenine-Moller 2001; Wald et al. 2003a] does not work for objects in which many polygons deform every frame. We believe that any practical real-time raytracing system must support moving and deformable objects with reasonable performance.

3 Integrate scene management with rendering

To ray trace dynamic scenes in real time we must reassess the crucial role of acceleration structures in making the ray tracing process efficient. The highest performance ray tracers use a space partitioning acceleration structure such as an octree or BSP tree, but the scene data is not originally stored in this form. Instead, the space partitioning data structure is constructed from data stored in a scene graph represented as a hierarchy of (potentially overlapping) bounding volumes.

A simple approach is to begin the computation of each frame by rebuilding an acceleration data structure of the type used in batch ray tracing. The problem with this approach is that the cost of rebuilding the acceleration structure may exceed the ray tracing cost itself. This problem is particularly serious if the scene has very high depth complexity, forcing the system to perform work for objects that are not hit by any rays. Even if we only rebuild those portions of the acceleration structure containing moving objects [Reinhard et al. 2000] the system may be performing much unnecessary work. For dynamic scenes it becomes apparent that minimizing the rebuild cost may be as important as minimizing the traversal cost, since the minimization of the total cost is the overriding criterion.

The most promising approach is to use lazy evaluation techniques to build the acceleration structure (building on and extending work by Ar *et al.* [Ar et al. 2002]). When a ray enters a previously untouched portion of the space partitioning data structure, the system puts the ray traversal on hold; then constructs that portion of the space partitioning data structure from the scene graph; and finally lets ray traversal resume through the newly created geometry.

However, this approach requires a close interaction between the acceleration data structure and the scene graph used to model the world at the application level. We believe that this recognition is the key to designing an effective system organization for real time dynamic ray tracing.

Consider a system in which scene management is tightly integrated with rendering (Figure 1). Such a system does not necessarily eliminate the need to store geometry using two different organizations – hierarchical scene graph and space partitioning – but such a system can tightly control which data is converted into the space partitioning form and when it is stored in this form. In particular,

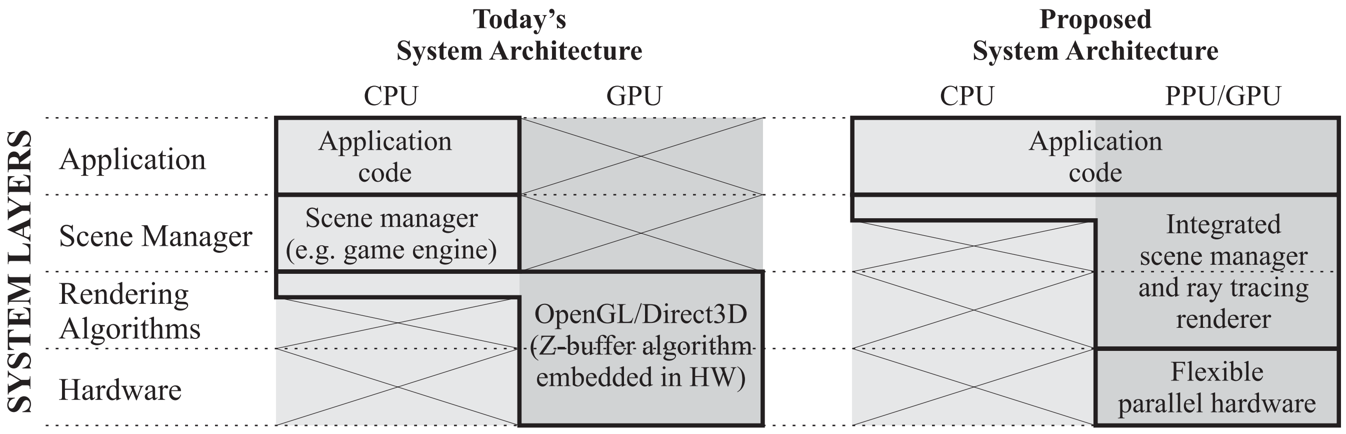
Today's System Architecture
Proposed System Architecture
CPU
GPU
CPU
PPU/GPU
SYSTEM LAYERS
Application
Scene Manager
Rendering Algorithms
Hardware
Application code
Scene manager (e.g. game engine)
OpenGL/Direct3D (Z-buffer algorithm embedded in HW)
Application code
Integrated scene manager and ray tracing renderer
Flexible parallel hardware

Figure 1: We propose that scene management be tightly integrated with rendering and that both be executed on the flexible parallel hardware. We refer to this flexible parallel hardware as a PPU (parallel processing unit).

the system can insure that only visible or nearly-visible surfaces are stored in space partitioning form.

Requiring the rendering system to integrate scene management with rendering is a major change from today's systems, so it is reasonable to ask why it is possible to separate scene management from rendering in a Z-buffer system but not in a ray tracer. In a simple Z-buffer system, visibility computations are performed in object order, so that each polygon in the scene is touched once and only once each frame by the visibility algorithm. Thus, for the purpose of the visibility computation, there is no need to store more than one polygon in local memory at a time. Of course the geometry must be stored somewhere in the system, but this can be done by the application or scene graph in any manner that is desired, with the geometry streamed across the immediate mode interface to the Z-buffer system. Commonly, the geometry is stored in a hierarchical data structure for the application to animate and otherwise modify.

Typically, ray tracing algorithms are "ray order" algorithms, in which the basic visibility algorithm can touch one polygon, then touch a second polygon, and eventually return to the first polygon. This type of algorithm requires direct access to the geometric database describing the scene. However, the geometric database need not be stored in the same format as the scene graph that is manipulated by the application. By transferring data lazily between the two data structures, we can minimize the cost of maintaining two different data structures.

3.1 Additional improvements

If ray traversal is managed so that most rays touching a particular portion of space are processed simultaneously [Pharr et al. 1997], then the system has the option of treating geometry represented in space-partitioning form as disposable. That is, when a particular volume of space is visited by a batch of rays, first the system creates an acceleration structure in on-chip memory for the geometry residing in that volume of space, then performs ray/triangle intersection tests, then discards the acceleration structure. The acceleration structure for that volume of space can be recreated later from the scene graph if it happens to be needed again.

Several other optimizations become convenient in this framework. If the system stores scene graph data using higher-order representations such as subdivision surfaces, these representations may be tesselated into triangles as the system creates the spatial acceleration structure. The data explosion that occurs during this step can be confined to on-chip memory, just as it is for a Z-buffer pipeline that includes a tesselation processor. Pharr and Hanrahan describe a variant of approach for displacement surfaces [Pharr and Hanrahan 1996].

An additional advantage of tight integration of scene management with rendering is that the system can automatically adapt the LOD of geometry to local ray density, even instantiating the same geometry at two different levels of detail, as is often needed when different types of rays (eye and reflected, for example) intersect the same geometry. A recent paper from Pixar [Christensen et al. 2003] has clearly demonstrated the value of using ray differentials to manage geometric level of detail in a raytracer.

4 Is a unified system organization practical?

We recognize that proposing to tightly couple scene management with rendering flies in the face of conventional wisdom about graphics system design. Current systems, following the lead of Iris GL and OpenGL [Segal and Akeley 2002], are characterized by the separation of scene management from rendering, mediated by a carefully-designed immediate mode rendering interface (Figure 1).

Why do we have this interface? Because experience has shown that it is not possible to build an efficient, fully general-purpose scene manager. Attempts to standardize systems of this type, such as CORE [Graphics standards planning committee 1979] and PHIGS [(american national standards institute) 1988], failed largely because of their attempt to integrate support for modeling and rendering using an API framework.

So why do we think we can do better? Because experience has also shown that it *is* possible to build reusable scene managers specialized for particular application domains. The most prominent examples are Performer [Rohlf and Helman 1994] which is specialized for visual simulation and id software's widely licensed game engines, which are specialized for first-person-shooter games. However, these systems do not use a standard API framework – either the engine is either highly configurable through internal hooks (Performer) or it can be directly modified in source code form (id's game engines).

We conclude that it is probably not possible to build a fully generic scene engine behind an API, but that it *is* possible to build specialized engines that implement performance critical tasks and can be adapted for particular applications. Thus, if one is willing to allow a scene manager to be implemented in "user" code (i.e. not embedded in unprogrammable hardware, or behind a one-size-fits-all interface), then it is perfectly possible to build a high-performance scene manager. If this scene manager can run on the same hardware that supports the rendering, then we believe that the scene manager can include rendering code, and thus provide the integrated renderer / scene manager that we propose. This approach is analogous to the programmable shaders in today's hardware, but carried much farther.

5 Parallel architecture supporting late binding

To efficiently support the ray tracing system we have described, the hardware architecture and corresponding parallel programming model must be very flexible and allow most control and data binding decisions to be deferred until run time. For example, a highly-specialized architecture, a SIMD architecture, or a streaming architecture would not be appropriate for this workload.

Several factors drive this need for generality:

- **Application-dependent scene management:** The architecture cannot be designed for particular scene management code.
- **Irregular data structures:** The scene graph and acceleration data structures are irregular, requiring pointer-chasing or its equivalent.
- **Dynamic data structures:** The irregular data structures must be built and modified with high performance, as well as being traversed with high performance.
- **Data dependent control flow:** Adaptive tesselation, ray tracing, and other tasks use highly data-dependent control flow.
- **Data locality:** Many of the data structures exhibit temporal locality in their access patterns, but the exact form of the locality is not known at compile time due to the irregular nature of the data structures.

We take as a starting point that our target architecture provides explicit parallelism, which provides better power efficiency than a single mainstream high-ILP CPU [Sasanka et al. 2004].

5.1 MIMD control flow

We advocate a MIMD programming model because it supports data parallel execution of computation kernels that use data-dependent conditionals and looping. Support for MIMD control flow is critical for efficiently creating and traversing adaptive spatial data structures such as k-d trees, as well as for executing short data-dependent loops such as those used in vertex skinning and anisotropic texture filtering [Sankaralingam et al. 2003a]. MIMD computation also supports general task level parallelism, i.e. it allows multiple distinct "kernels" to run concurrently. A primary example of the need for this is to allow closely coupled scene graph management and rendering tasks to run concurrently, particularly when these tasks are not individually sufficiently parallelizable to be able to occupy the entire machine.

Current graphics hardware (e.g. NVIDIA 6800 with shader model 3.0) supports a more restrictive SPMD (single-program, multiple data) programming model in which MIMD-style control flow is supported, but all fragment or vertex processors must be running the same program. However, the hardware implementation of the control flow is closer to a SIMD implementation, so that code with divergent branching behavior is inefficient [Nvidia Corp. 2004].

Even if future architectures use a MIMD organization as we advocate, that does not preclude support for simpler programming models as well. Most other parallel programming models (e.g. various variants of "stream programming") can be described as restricted subsets of the one we have outlined and thus can be supported by the same hardware. For tasks that can tolerate these limitations, the restricted programming models are often easier to use and typically encourage the programmer to express the task in a form that will perform well. For example, the stream programming model forbids the code within one kernel from directly communicating with the code within another kernel, thereby eliminating the potential for many types of concurrency and performance bugs.

Recent industry designs seem to endorse our view that MIMD architectures are a better choice than SIMD architectures for general-purpose single-chip parallel computation. Sun's Niagara [Krewell 2003] and IBM/STI's CELL [Pham et al. 2005] are both fully MIMD. The most advanced graphics processors (e.g. GeForce 6800) currently have a MIMD programing model (actually SPMD) implemented as a MIMD execution model in the vertex processor and a SIMD execution model in the fragment processor. We expect future architectures to gradually move towards a MIMD implementation, although maintaining current fragment ordering semantics in a MIMD machine presents some challenges. Several interesting research architectures that use a highly-parallel MIMD organization are IBM's Cyclops [Caşcaval et al. 2002] (not yet built), Stanford's Smart Memories [Mai et al. 2000] (not yet built), MIT's RAW [Taylor et al. 2004] (already built), and the MIT M-Machine [Keckler et al. 1998] (already built) which demonstrated some promising approaches for supporting fine-grained MIMD parallelism.

Note that although all of the architectures mentioned above are MIMD in their overall organization, many of them support 4-wide SIMD instructions within each core. These short-vector SIMD instructions are an efficient mechanism for exploiting what is really just a particularly common form of instruction-level parallelism found in graphics and scientific code. Even in machines that are designed to exploit MIMD thread-level parallelism, it is still worthwhile to support such low-cost forms of instruction level parallelism, since exploiting such parallelism improves performance without requiring an increase in on-chip storage such as would be required by support for additional threads.

5.2 Hardware caches and global address space

For processors built using modern VLSI technology it is desirable to include a multi-level memory hierarchy on chip, since for workloads with temporal memory-access locality this strategy provides a favorable combination of low power consumption, low average memory-access latency, and high load/store bandwidth [Kapasi et al. 2002].

There are a variety of mechanisms by which a programming model can provide access to high-speed on-chip memory. The two most popular mechanisms are a hardware-managed cache and a software-managed scratchpad memory. The difference between these two approaches is fundamental. For a cache, the decision as to which elements of data should be stored on chip is automatically made by the hardware at run-time, with the decision typically made at a fine granularity (e.g. blocks of 32 bytes). With a software-managed scratchpad memory, the decision as to which data should be stored on chip is made either at compile time or made explicitly by software at runtime, usually at a coarser granularity.

In applications with highly regular memory access patterns, such as classical DSP applications, a software-managed memory is the right choice. Software-managed memories carry less hardware overhead, allow static scheduling of the entire machine (particularly important for SIMD architectures), and provide the user and compiler with better performance guarantees than a hardware-managed cache.

In contrast, applications that manipulate adaptive data structures such as k-d trees, BSP trees, or short variable-length lists cannot easily manage memory at compile time. The application writer and compiler may know that there will be significant spatial and temporal locality of the memory accesses, but they do not know exactly what form this locality will take for any particular data set. For these applications, the binding of particular data elements to the on-chip memory is best performed at a fine spatial granularity. This approach is exactly that used by conventional hardware-managed caches. Since we believe that the construction, modification, and use of adaptive spatial data structures will be a performance-critical part of future real-time 3D graphics applications, we believe that future hardware architectures should support hardware-managed caches or at a minimum must include hardware primitives from which equivalent behavior can be efficiently implemented in software.

One important advantage of an architecture with traditional hardware-managed caches – especially if cache-coherency is supported – is that the architecture can provide the illusion of a single large memory, in which the storage hierarchy is simply an automatic hardware-supported performance optimization. In practice, parallel software must be heavily tuned to achieve good performance from such an architecture, but this performance tuning can be done incrementally. In contrast, software-managed memories are usually exposed to the programmer and/or compiler as a series of architecturally visible capacity "cliffs", which must be painfully overcome even in the earliest software prototypes.

The recently announced CELL architecture [Pham et al. 2005], is an interesting hybrid between the traditional cache strategy and scratchpad strategy. CELL's parallel cores (called SPE's) have a local scratchpad memory, but the DMA transfers between this scratchpad and main memory are coherent within a single global address space. The difficulty of managing a scratchpad memory is mitigated by the fact that the scratchpad is unusually large (256 KB per core). For a programmer, is is qualitatively easier to manage this L2-sized scratchpad than it is to manage a more traditional L1-sized scratchpad. Nevertheless, we believe that it will prove to be challenging to efficiently implement some irregular-datastructure algorithms on CELL. Even the strategy of using software to mimic traditional cache behavior is unlikely to perform well on CELL, due to the long branch mis-predict penalty and lack of hardware-supported multithreading. However, we believe that adding minimalist multithreading capability to the CELL SPE architecture would substantially improve this situation at relatively low cost, and we hope that this capability will be considered for future versions of CELL.

5.3 Hardware support for multithreading

A major problem encountered by most modern architectures is that the latency for moving data between the processing chip and off-chip DRAM memory can be 100 or more cycles. To maintain high ALU utilization, the machine must perform other work while such requests are outstanding. With a hardware-managed cache, the problem is particularly severe, because the compiler and hardware do not know in advance whether a particular 'load' or 'store' will miss the cache(s). Thus, *every* access to the unified address space potentially incurs a 100 cycle delay, whereas in a machine with a scratchpad, only the explicit accesses to off-chip memory can incur this delay.

Fortunately, highly parallel computations such as those in 3D graphics normally have other work (i.e. other threads) that can be processed during an off-chip memory access. There are two strategies for switching to other thread(s), which we will now describe.

The first strategy is to assume that every memory access misses the cache. This approach is followed by classical texture caching systems [Igehy et al. 1998], by the specialized SaarCOR raytracing architecture [Schmittler et al. 2004], and by cacheless multithreaded architectures like Tera [Alverson et al. 1990]. The ALU switches to other thread(s) (e.g. another fragment or vertex) for the required number of cycles, regardless of whether or not the memory request actually missed the cache. Unfortunately, this strategy requires that the number of active threads per ALU be approximately equal to the off-chip memory latency. The memory needed to store the working set for these threads can easily dominate the die area of the parallel processor, particularly when one considers the data-cache or scratchpad-memory footprint of each thread as well as its registers.

The second strategy is to switch to another thread only if the data access actually misses the cache. This approach is the one used by modern multithreaded machines such as Niagara [Krewell 2003], Cyclops [Caşcaval et al. 2002], and MAJC [Kowalczyk et al. 2001]. The advantage of this second approach is that fewer threads are required, particularly if cache misses are infrequent. Thus we consider this strategy to be the better one, at least if the machine is already a MIMD machine. However, it is worth noting that this strategy may not perform well if the memory accesses by different threads are highly correlated, leading to situations where all threads stall at the same time waiting for the same cache line. For example, this situation can occur for texture map lookups in a fragment shader. In some cases careful use of 'prefetching' can mitigate this problem, but it is not yet clear if this strategy would be effective for texture mapping.

There is an unfortunate tension between the goal of maximizing overlap of the working sets of different threads (which in turn reduces the per-thread SRAM requirements) and minimizing the temporal correlation between cache misses of different threads (which in turn allows a reduction in the ratio of threads-per-ALU). We expect that managing this tradeoff will be a major focus of future performance-optimization efforts for both hardware and software in single-chip parallel systems. One advantage of SIMD control flow that is often under-appreciated is that SIMD execution provides implicit but very tight inter-thread synchronization that facilitates reasoning about and management of this tradeoff. Managing this tradeoff in MIMD systems can require that fine-grained inter-thread synchronization be used for this purpose as well as for the traditional purpose of managing the more obvious control and data dependencies in the parallel computation.

5.4 Parallelism Summary

The various design decisions for a parallel machine are closely coupled to each other. For example, the decision to use a hardware-managed L1 cache in each core is at odds with a decision to use SIMD control. Broadly speaking, there appear to be two reasonable points in the design space, which can be referred to as "static" and "dynamic". Static machines such as Imagine bind and schedule most fine-grain resources at compile time – ALUs, on-chip memory, off-chip memory accesses, etc. The static strategy can use compile-time information about the program, but cannot not use much if any information about data-dependent behavior. In contrast, dynamic machines such as Niagara [Krewell 2003], Cyclops [Caşcaval et al. 2002] and the Intel IXP network processor [Adiletta et al. 2002] bind and schedule most resources at run time with hardware assistance. The dynamic-binding strategy uses both program information and runtime information derived from the data being processed.

For tasks in which the runtime information can significantly improve the quality of resource binding and scheduling, we believe that the dynamic approach will provide superior performance and will also be easier to program. However, for problems that can be effectively scheduled at compile time, there is no benefit to the dynamic approach, and the hardware support needed for it reduces the performance/price ratio of the hardware. Thus, the decision as to what type of machine to build should rest largely on anticipated application characteristics. We have argued that future real-time 3D graphics algorithms will use adaptive data structures, and thus that future architectures targeted to support these algorithms should use dynamic binding and scheduling. The close coupling we find here between the choice of software algorithms and the choice of hardware architectures is one of the reasons that we are advocating that algorithmic and hardware questions be investigated in tandem.

As with most such design-space tradeoffs, hybrid strategies exist. For example CELL has MIMD control flow, seemingly placing it in the dynamic category, but its high branch-mispredict penalty coupled with lack of multithreading somewhat penalize highly dynamic algorithms, as does CELL's choice of scratchpad memory rather than cache for local storage. A useful perspective on the general static-vs-dynamic tradeoff can be found in the architectural taxonomy found at the end of [Taylor et al. 2004].

5.5 CPU and parallel processor on the same die

Experience teaches us that very few problems are perfectly parallelizable. Historically, Cray's vector machines outperformed their competitors because they had superior performance for scalars and short vectors [Hennessy et al. 2003]. 3-D graphics is no exception to this general rule – modern graphics hardware has serialization points, although these potential bottlenecks are normally not user programmable.

For this reason, we believe that future graphics algorithms will split their work between an array of parallel processors optimized for high, power-efficient throughput on parallel code and at least one CPU-like processor optimized for maximum performance on a single thread. We believe that these two core types will be implemented with different sets of transistors, rather than by reconfiguring a single underlying substrate [Sankaralingam et al. 2003b; Taylor et al. 2004; Mai et al. 2000]. The reason is that a well-designed throughput-optimized processor differs from a single-thread-optimized processor in almost every respect, including the physical design of the individual transistors. The flexibility gained from a single reconfigurable substrate is likely to be more than offset by the cost of the necessary compromises.

To facilitate the low-latency, high-bandwith transfer of work between the throughput-optimized and single-thread-optimized processing cores, they must be integrated on a single die. Network processors [Adiletta et al. 2002] and CELL use this organization already, and we believe that in the long term these technical benefits as well as market trends towards cost reduction make such integration inevitable for graphics processors.

5.6 More than one kind of throughput-optimized core?

One important but open question is whether future chip-multiprocessors should have just one kind of throughput-optimized core, or two or more varieties of such cores. For example, it would be reasonable to build an architecture which has one set of cores that can only write to memory via stream outputs (like today's GPU fragment processors), and a second set of cores supporting cache-coherent memory writes and reads. The first set of cores would have higher peak performance, but would be restricted to a narrower class of computations than the second set of cores.

Other kinds of cores may also be useful. For example, current graphics chips include a simple configurable hardware unit (the raster-operation unit) located next to each of several memory controllers. We have shown that adding additional capabilities to this unit enables it to efficiently assist the task of building linked lists [Johnson et al. 2005]. Others have shown that such "near-memory processing" can be useful for traversing linked lists [Hughes and Adve 2005].

Finally, if a single-chip parallel architecture is expected to be heavily used for one particular task such as 3D rendering, it may be advantageous to include highly-specialized cores optimized for particular tasks such as texture filtering (included in today's GPU's) or ray/triangle intersection testing.

Most of these decisions must be made based on detailed cost/benefit analysis of both the workload and the hardware implementation, but there is one broad issue that will impact all such decisions. It is possible that future power budgets will prohibit architectures from using all of their transistors at once. This constraint would favor heterogeneous specialization of the architecture's processing units, a point that was first brought to our attention by Mark Horowitz.

6 Conclusion

We have argued that the next frontier in improved real-time image quality is to simulate global illumination effects for dynamic scenes. We claim that ray tracing will be the visibility algorithm of choice, but that it will initially be used to support non-physically-correct global illumination techniques.

We believe that a ray tracing system that efficiently supports dynamic scenes will need to integrate scene management with rendering. Since scene management code is somewhat application specific, this tight integration implies that the parallel architecture used to accelerate rendering must also be capable of executing application-specific scene management code. In turn, this requires that the parallel architecture support a general-purpose parallel programming model, with inter-thread communication, synchronization, and perhaps cache-coherent memory operations. The programming model supported by today's GPUs lacks most of these capabilities, and in particular it does not provide adequate support for creating and modifying adaptive data structures.

We believe the most promising hardware architecture to support this programming model is a MIMD multithreaded machine with cache-coherent shared memory. However, this conjecture remains unproven, and many questions remain about the details of such an architecture as well as its price/performance ratio.

To date we have not built either the software or the hardware necessary to confirm our hypothesis. What we have presented is a set of informed opinions backed by reasonable arguments and some initial results from architecture and algorithm simulations [Johnson et al. 2005]. Our purpose in presenting these opinions is two-fold. First, we think the ideas are sufficiently interesting that they will stimulate useful discussion within the research community. Second, we hope to persuade the research community that the particular approach we have outlined is sufficiently promising to be worthy of detailed investigation.

7 Acknowledgements

Gordon Stoll has contributed substantially to our thinking about real-time ray tracing systems and in particular is responsible for the insight that ray casting visiblity is likely to be used initially to support non-physically-correct techniques such as ambient occlusion rather than to support physically-correct techniques such as photon mapping.

Kurt Akeley is responsible for the insight that scene-graph rendering interfaces can be successful if they are tailored to sufficiently narrow application domains.

Our research group and collaborators – Greg Johnson, Paul Navratil, Calvin Lin, Chris Burns, Juhyun Lee, Karthikeyan Sankaralingam, Doug Burger, Steve Keckler, the Stanford Smart Memories group, Alex Joly, Ikrima Elhassan, David Whiteford, Chris Lundberg, Chendi Zhang and Peter Djeu – have been building systems and gathering results that have influenced our thinking about both rendering algorithms and parallel architectures.

We thank Microsoft, Intel, NVIDIA, and the University of Texas for supporting this research.

References

ADILETTA, M., ROSENBLUTH, M., BERNSTEIN, D., WOLRICH, G., AND WILKINSON, H. 2002. The next generation of Intel IXP network processors. *Intel technology journal 6*, 3 (Aug.).

AKELEY, K. 1993. RealityEngine graphics. In *SIGGRAPH 93*, 109–116.

ALVERSON, R., CALLAHAN, D., CUMMINGS, D., KOBLENZ, B., PORTERFIELD, A., AND SMITH, B. 1990. The tera computer system. *SIGARCH Comput. Archit. News 18*, 3, 1–6.

(AMERICAN NATIONAL STANDARDS INSTITUTE), A. 1988. programmer's hierarchical interactive graphics system (PHIGS) functional description. Tech. rep., ANSI.

AR, S., MONTAG, G., AND TAL, A. 2002. Deferred, self-organizing bsp trees. In *Eurographics 2002*.

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRODER, P. 2003. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *SIGGRAPH 2003*.

CAŞCAVAL, C., NOS, J. G. C., CEZE, L., DENNEAU, M., GUPTA, M., LIEBER, D., MOREIRA, J. E., STRAUSS, K., AND JR, H. S. W. 2002. Evaluation of a multithreaded architecture for cellular computing. In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, IEEE Computer Society, 311–322.

CHRISTENSEN, P. H., LAUR, D. M., FONG, J., WOOTEN, W. L., AND BATALI, D. 2003. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. In *Eurographics 2003*.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The REYES image rendering architecture. *SIGGRAPH 87 21*, 4 (July), 95–102.

GRAPHICS STANDARDS PLANNING COMMITTEE. 1979. Status report of the graphics standards planning committee. *Computer graphics 13*, 3 (Aug.).

GRITZ, L., AND HAHN, J. K. 1996. BMRT: A global illumination implementation of the RenderMan standard. *Journal of Graphics Tools 1*, 3, 29–47.

HANRAHAN, P., AND LAWSON, J. 1990. A language for shading and lighting calculations. In *SIGGRAPH 90*, 289–298.

HENNESSY, J. L., PATTERSON, D. A., AND GOLDBERG, D. 2003. *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann.

HUGHES, C. J., AND ADVE, S. V. 2005. Memory-side prefetching for linked data structures for processor-in-memory systems. *Journal of Parallel and Distributed Computing 65*, 4 (Apr.), 448–463.

HURLEY, KAPUSTIN, RESHETOV, AND SOUPIKOV. 2002. Fast ray tracing for modern general purpose CPU. In *Graphicon 2002*.

HURLEY, J., 2005, Mar. Personal Communication.

IGEHY, H., ELDRIDGE, M., AND PROUDFOOT, K. 1998. Prefetching in a texture cache architecture. In *Proc. of 1998 Eurographics/SIGGRAPH workshop on graphics hardware*.

JOHNSON, G. S., LEE, J., BURNS, C. A., AND MARK, W. R. 2005. The irregular z-buffer. *ACM Transactions on Graphics (to appear)*.

KAPASI, U. J., DALLY, W. J., RIXNER, S., OWENS, J. D., AND KHAILANY, B. 2002. The Imagine stream processor. In *Proc. of IEEE Conf. on Computer Design*, 295–302.

KATO, T. 2002. The "Kilauea" massively parallel ray tracer. In *Practical Parallel Rendering*, A K Peters, A. Chalmers, T. Davis, and E. Reinhard, Eds.

KECKLER, S. W., DALLY, W. J., MASKIT, D., , CARTER, N. P., CHANG, A., AND LEE, W. S. 1998. Exploiting fine-grain thread level parallelism on the MIT multi-alu processor. In *ISCA 1998*, 306–317.

KELLER, A. 1997. Instant radiosity. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 49–56.

KOWALCZYK, A., ADLER, V., AMIR, C., CHIU, F., CHNG, C. P., LANGE, W. J. D., GE, Y., GHOSH, S., HOANG, T. C., HUANG, B., KANT, S., KAO, Y. S., KHIEU, C., KUMAR, S., LEE, L., LIEBERMENSCH, A., LIU, X., MALUR, N. G., MARTIN, A. A., NGO, H., OH, S.-H., ORGINOS, I., SHIH, L., SUR, B., TREMBLAY, M., TZENG, A., VO, D., ZAMBARE, S., AND ZONG, J. 2001. The first majc microprocessor: A dual cpu system-on-a-chip. *IEEE Journal of Solid-State Circuits 36*, 11 (Nov.), 1609–1616.

KREWELL, K. 2003. Sun weaves multithreaded future. Available online at http://www.sun.com/processors/throughput/MDR_Reprint.pdf.

LANDER, J. 1998. Skin them bones: game programming for the web generation. *Game Developer Magazine* (May), 11–16.

LEXT, J., AND AKENINE-MOLLER, T. 2001. Towards rapid reconstruction for animated ray tracing. In *Eurographics 2001*.

LINDHOLM, E., KILGARD, M. J., AND MORETON, H. 2001. A user-programmable vertex engine. In *SIGGRAPH 2001*.

MAI, K., PAASKE, T., JAYASENA, N., HO, R., DALLY, W. J., AND HOROWITZ, M. 2000. Smart memories: A modular reconfigurable architecture. In *ISCA 2000*.

MARK, W. R., GLANVILLE, S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: A system for programming graphics hardware in a C-like language. In *SIGGRAPH 2003*.

MOYER, B., 2004. Ambient occlusion: It's better than a kick to the head. WWW page visited December 2004, http://www-viz.tamu.edu/students/bmoyer/617/ambocc/.

NVIDIA CORP. 2003. NV_fragment_program. In *NVIDIA OpenGL Extension Specifications*. Jan.

NVIDIA CORP. 2004. *NVIDIA GPU programming guide, v2.2.1*, Nov.

PARKER, S., MARTIN, W., SLOAN, P.-P. J., SHIRLEY, P., SMITS, B., AND HANSEN, C. 1999. Interactive ray tracing. In *Symposium on interactive 3D graphics*.

PARKER, S., PARKER, M., LIVNAT, Y., SLOAN, P.-P., HANSEN, C., AND SHIRLEY, P. 1999. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics 5*, 3, 238–250.

PHAM, D., S.ASANO, BOLLIGER, M., DAY, M., HOFSTEE, H., JOHNS, C., KAHLE, J., KAMEYAMA, A., KEATY, J., MASUBUCHI2, Y., RILEY1, M., SHIPPY1, D., STASIAK1, D., M.WANG, J.WARNOCK, S.WEITZEL, D.WENDEL, T.YAMAZAKI, AND K.YAZAWA. 2005. The design and implementation of a first-generation cell processor. In *Proceedings of 2005 IEEE Intl. Solid-State Circuits Conf.*

PHARR, M., AND HANRAHAN, P. 1996. Geometry caching for ray-tracing displacement maps. In *1996 Eurographics workshop on rendering*.

PHARR, M., KOLB, C., GERSHBEIN, R., AND HANRAHAN, P. 1997. Rendering complex scenes with memory-coherent raytracing. In *SIGGRAPH 1997*.

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. In *SIGGRAPH 2002*, 703–712.

REINHARD, E., SMITS, B., AND HANSEN, C. 2000. Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the 11th Eurographics Workshop on Rendering*, Eurographics Association, 299–306.

ROHLF, J., AND HELMAN, J. 1994. IRIS performer: A high performance multiprocessing toolkit for real–time 3D graphics. In *SIGGRAPH 94*, 381–394.

SANKARALINGAM, K., KECKLER, S. W., MARK, W. R., AND BURGER, D. 2003. Universal mechanisms for data-parallel architectures. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*.

SANKARALINGAM, K., NAGARAJAN, R., LIU, H., KIM, C., HUH, J., BURGER, D., KECKLER, S. W., AND MOORE, C. R. 2003. Exploiting ilp,tlp, and dlp with the polymorphous trips architecture. In *Proc. of the 30th Annual Intl. Symp. on Computer Architecture (ISCA)*.

SASANKA, R., ADVE, S. V., CHEN, Y.-K., AND DEBES, E. 2004. The energy efficiency of cmp vs. smt for multimedia workloads. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, ACM Press, New York, NY, USA, 196–206.

SCHMITTLER, J., WOOP, S., WAGNER, D., PAUL, W. J., AND SLUSALLEK, P. 2004. Realtime ray tracing of dynamic scenes on an fpga chip. In *Graphics Hardware 2004*.

SEGAL, M., AND AKELEY, K. 2002. *The OpenGL Graphics System: A Specification (Version 1.4)*. OpenGL Architecture Review Board. Editor: Jon Leech.

SLOAN, P.-P., KAUTZ, J., AND SNYDER, J. 2002. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 527–536.

TAYLOR, M. B., LEE, W., MILLER, J., WENTZLAFF, D., BRATT, I., GREENWALD, B., HOFFMANN, H., JOHNSON, P., KIM, J., PSOTA, J., SARAF, A., SHNIDMAN, N., STRUMPEN, V., FRANK, M., AMARASINGHE, S., AND AGARWAL, A. 2004. Evaluation of the raw microprocessor: An exposed-wire-delay architecture for ilp and streams. In *ISCA 2004*.

THOMPSON, C. J., HAHN, S., AND OSKIN, M. 2002. Using modern graphics architectures for general-purpose computing: a framework and analysis. In *Intl. symposium on computer architecture*.

WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. Distributed interactive ray tracing of dynamic scenes. In *Proc. IEEE symp. on parallel and large-data visualization and graphics*.

WALD, I., PURCELL, T. J., SCHMITTLER, J., BENTHIN, C., AND SLUSALLEK, P. 2003. Realtime ray tracing and its use for global illumination. In *Eurographics 2003*.

WALD, I., PURCELL, T. J., SCHMITTLER, J., BENTHIN, C., AND SLUSALLEK, P. 2003. Realtime ray tracing and its use for interactive global illumination. In *State of the Art Reports, EUROGRAPHICS 2003*.

WANN JENSEN, H. 2001. *Realistic image synthesis using photon mapping*. AK Peters.

WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (June), 343–349.