The Irregular Z-Buffer: Hardware Acceleration for Irregular Data Structures

GREGORY S. JOHNSON, JUHYUN LEE, CHRISTOPHER A. BURNS, and WILLIAM R. MARK
University of Texas at Austin

The classical Z-buffer visibility algorithm samples a scene at regularly spaced points on an image plane. Previously, we introduced an extension of this algorithm called the *irregular Z-buffer* that permits sampling of the scene from arbitrary points on the image plane. These sample points are stored in a two-dimensional spatial data structure. Here we present a set of architectural enhancements to the classical Z-buffer acceleration hardware which supports efficient execution of the irregular Z-buffer. These enhancements enable efficient parallel construction and query of certain irregular data structures, including the grid of linked lists used by our algorithm. The enhancements include flexible atomic read-modify-write units located near the memory controller, an internal routing network between these units and the fragment processors, and a MIMD fragment processor design. We simulate the performance of this new architecture and demonstrate that it can be used to render high-quality shadows in geometrically complex scenes at interactive frame rates. We also discuss other uses of the irregular Z-buffer algorithm and the implications of our architectural changes in the design of chip-multiprocessors.

Categories and Subject Descriptors: I.3.1 [**Computer Graphics**]: Hardware Architecture—*Graphics processors*; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Visible line/surface algorithms*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Real-time graphics hardware, shadow algorithms, visible surface algorithms, architecture, computer graphics

1. INTRODUCTION

One of the fundamental computational tasks in three-dimensional graphics is the visible surface problem: to efficiently find the first intersection point between a ray and a collection of surfaces. This problem is typically solved by either the Z-buffer algorithm [Catmull 1974], or by ray tracing [Appel 1968; Whitted 1980]. Nearly all modern real-time graphics systems use the Z-buffer, which is supported by specialized hardware.

The algorithms used in real-time rendering are tightly coupled to the graphics architectures that support them. The overall organization of these architectures have changed surprisingly little over the past 15 years. As a result, the Z-buffer continues to prevail despite experiences indicating that more flexible visibility algorithms would yield higher-quality images [Cook et al. 1987; Tabellion and Lamorlette 2004] and would be adopted if they could be implemented with adequate performance.

This work was supported by Microsoft, an NVIDIA graduate fellowship, NSF, and an Intel equipment grant.
Authors' addresses: G. Johnson, C. Burns, Texas Advanced Computing Center, 10100 Burnet Rd. (R8700), University of Texas at Austin, Austin, TX 78758-4497; email: {johnsong,cburns}@cs.utexas.edu; W. Mark, J. Lee, Department of Computer Sciences, Taylor Hall 2.124, University of Texas at Austin (CO 500), Austin, TX 78712-0233; email: {impjdi,billmark}@cs.utexas.edu.

The irregular Z-buffer provides some of the flexibility of ray tracing while largely preserving the performance characteristics of the classical Z-buffer. The irregular Z-buffer solves the visible surface problem for rays sharing a single origin, but with arbitrary directions [Johnson et al. 2004; Aila and Laine 2004]. It is thus more general than the classical Z-buffer in which ray directions must follow a regular pattern. However, the irregular Z-buffer is not as general as ray tracing. In ray tracing both the direction *and* origin of rays are arbitrary, permitting visibility testing between any two points in a scene.

The irregular Z-buffer also shares the system organization of the classical Z-buffer. Both are immediate-mode object-order renderers, enabling hardware designed for the latter to be used (with the modifications discussed later) for the former. In contrast, ray tracing operates in image-order. A spatial acceleration structure is often used to reduce the number of ray-object intersection tests. However, efficient construction of these data structures on classical Z-buffer architectures is an open problem.

As in the classical Z-buffer, the irregular Z-buffer represents ray directions as points on an image plane. However, since these samples can be arbitrarily placed, their locations are explicitly stored in a two-dimensional spatial data structure. During rasterization, triangles are projected onto the image plane as usual. The data structure is then queried to determine which samples are overlapped by the projected triangles. Finally, a standard Z-compare and conditional update operation is performed for each overlapped sample.

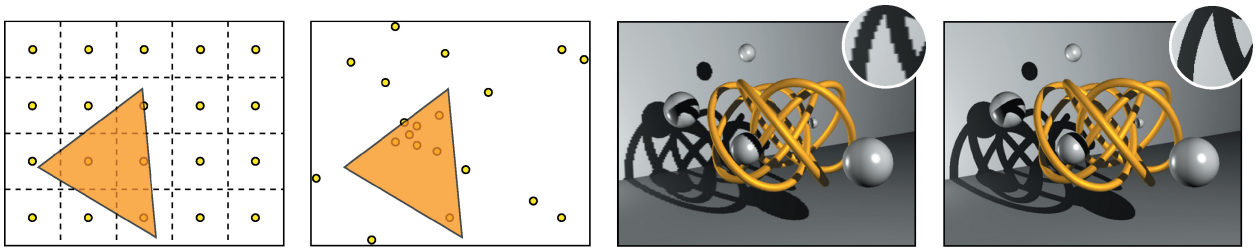
The principal contribution of this work is the design and analysis of a set of architectural enhancements necessary for hardware acceleration of the irregular Z-buffer algorithm. The restrictions imposed by the classical Z-buffer architecture have become increasingly apparent since the introduction of programmability [Lindholm et al. 2001]. In particular, current graphics architectures lack efficient support for scatter operations and for most forms of read-modify-write memory operations. Our work shows that the memory access flexibility enabled by these operations (along with other enhancements) allows efficient execution of the irregular Z-buffer algorithm on an architecture derived from current designs.

More broadly, our work demonstrates that it is possible to design a parallel graphics architecture with efficient support for the creation and traversal of irregular data structures. In particular, the irregular Z-buffer uses a grid of variable-length linked lists to represent the locations of sample points on an image plane. Irregular data structures such as these are used throughout graphics. Adding hardware support for them significantly widens the class of algorithms which are able to run efficiently on GPUs. Moreover, we believe graphics architectures are on a convergent course with general-purpose chip-multiprocessors (CMPs). As such, several features of our design could be used to enhance the performance of CMPs in the area of irregular data structure construction and query.

The following sections review the irregular Z-buffer algorithm introduced by Johnson et al. [2004] and Aila and Laine [2004], and illustrate its use in rendering high-quality hard shadows in real time. Our supporting architecture is described in Section 4, with a focus on the novel aspects of the design. Simulation results for this architecture (Section 5) demonstrate that it is capable of rendering shadows via the irregular Z-buffer algorithm at interactive frame rates. Several performance issues, design choices, and other applications of the irregular Z-buffer are explored in Section 6.

2. THE IRREGULAR Z-BUFFER

The classical rasterization algorithm projects each polygon onto the image plane, and determines which sample points from a regularly spaced set lie inside the projected polygon. Since the locations of these samples (i.e., pixels) are implicit, this determination can be made by testing the edges of the polygon against the implicit grid of sample points. If, however, the locations of the sample points are irregularly spaced and cannot be computed from a formula, then this approach does not work (see Figures 1(a) and 1(b)). The irregular Z-buffer solves this problem by storing sample locations explicitly



(a) Conventional Z-buffer (b) Irregular Z-buffer (c) Conventional shadow mapping (d) Irregular shadow mapping

Fig. 1. The classical Z-buffer (a) samples a scene at regularly spaced points on an image plane. The irregular Z-buffer (b) samples a scene at arbitrary points on an image plane. This capability has many applications including shadow rendering, where it eliminates (d) the self-shadowing and aliasing artifacts typically associated with conventional shadow mapping (c).

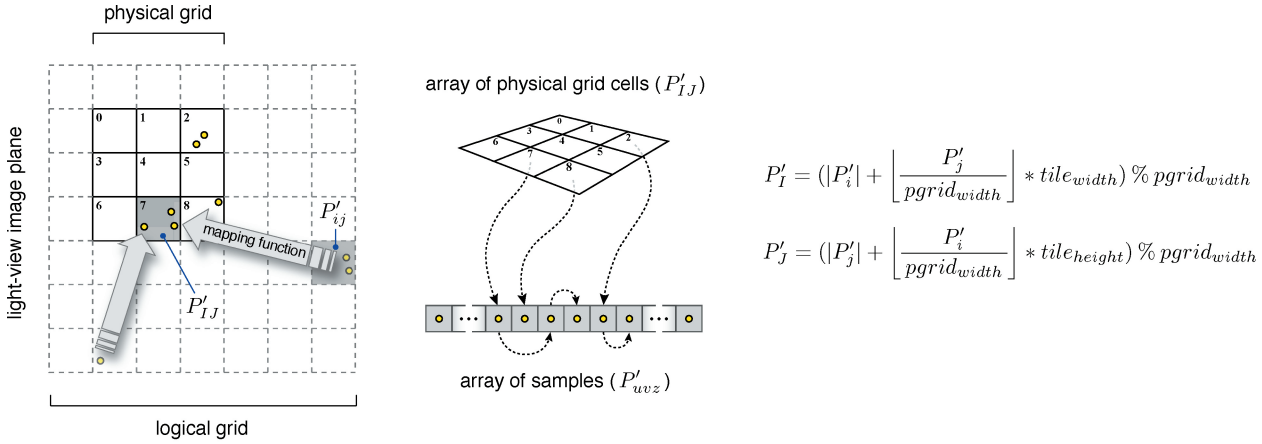
in a two-dimensional spatial data structure, and later querying this structure to determine which samples lie within the projected triangle. We refer to the latter step as *irregular rasterization*.

The spatial data structure may take one of several forms. Candidates include variants of the BSP tree (i.e., quadtrees and k-d trees), and regular grids. The best choice depends on the intended use. For applications where the set of sample points changes significantly from frame to frame, the cost of construction is as important as query cost. These costs can vary asymptotically between data structures, and may also depend on the distribution of points or the order in which they are inserted. Finally, the storage requirements may depend on the point distribution, leading to a variable memory footprint from frame to frame. This should be avoided. We explore storage-related issues further in Section 6.5.

Our data structure is a hybrid composed of a grid with linked lists at each grid cell. This data structure is similar to the grid of variable-lengthed arrays used by Purcell and colleagues in their GPU implementation of ray tracing [Purcell et al. 2002] and photon mapping [Purcell et al. 2003]. In contrast, Aila and Laine [2004] employed a k-d tree. We favor a grid-based structure over trees for several reasons. First, the selection of a grid as the primary structure permits the use of classical rasterization hardware during the early stages of the irregular rasterization process. Second, the point-in-area test at each step of list traversal is currently less expensive than the area-to-area overlap test required at each step of k-d tree traversal. Third, current GPU instruction sets [Brown and Werness 2004] favor list traversal over tree traversal in terms of the number of instructions required to traverse one node, due to the limited capability of current branching instructions. We anticipate the suitability of any given structure will change with the evolution of graphics architectures.

The grid portion of our data structure consists of a large *logical* grid spanning the image plane, and a smaller stored *physical* grid. Cells in the logical grid are mapped into the physical grid, reducing the memory footprint of this portion of the data structure [Reinhard et al. 2000]. The two grids and the function which maps between them are shown in Figure 2. Our mapping function has several key properties. It avoids folding different parts of a given region of high sample density in the logical grid onto the same cell(s) in the physical grid. This property inhibits further concentration of samples in hot spots in the logical grid during transformation into the physical grid. The mapping also preserves a high degree of spatial coherence. Points nearby in the logical grid tend to remain nearby in the physical grid. This coherence is exploited during irregular rasterization by tiling the physical grid in memory and processing fragments in tile order [McCormack et al. 1998].

The need to explicitly store sample locations in a spatial data structure results in a two-phase Z-buffer algorithm. In the first phase, sample locations are computed and stored in the data structure. In the second, triangles are rasterized to the logical grid, the resulting fragments are mapped into the physical



(a) Irregular Z-buffer data structure (b) Data structure storage (c) Logical-to-physical grid mapping function

Fig. 2. (a) The data structure used in the irregular Z-buffer algorithm. (a) The image plane is overlaid with a logical grid. The logical grid has a higher resolution than the stored (physical) grid. (b) Each physical grid cell potentially serves as the root of a linked list. (c) The linked list contains the samples which map to a given cell, for later use during rasterization. Logical grid cell indices (P'_{ij}) are mapped to physical grid cell indices (P'_{IJ}) via the function shown in (c). The $tile$ terms relate to tiling of the data structure in memory for improved cache reuse, and $pgrid_{width}$ denotes the dimensions of the physical grid.

grid, and the corresponding linked lists of samples are traversed. These two phases are detailed in the next section in the context of shadow rendering.

3. SHADOW RENDERING

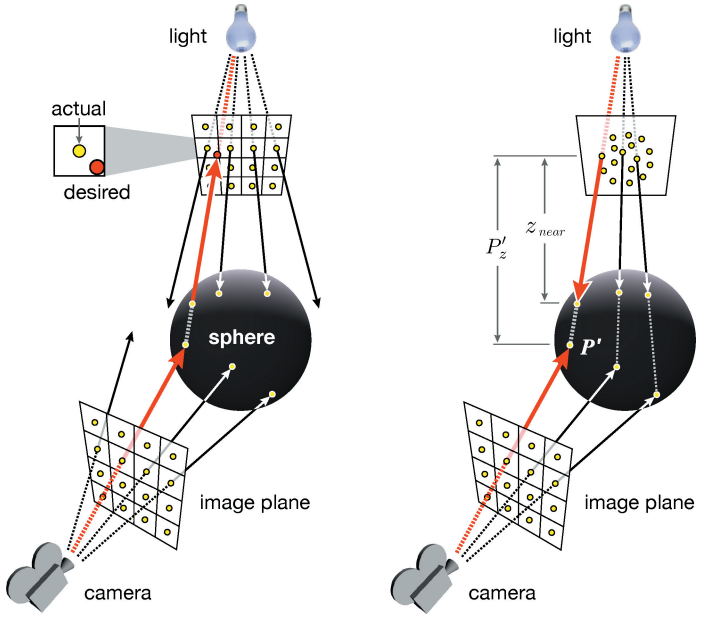
Like the classical Z-buffer, the irregular Z-buffer is a general-purpose visibility algorithm and can thus be used for a wide variety of tasks. Our work focuses primarily on rendering hard shadows in real-time for scenes containing moving and deformable objects. Even though the problem is conceptually simple and is of practical interest, existing approaches have significant shortcomings with respect to image quality or performance.

Here, we illustrate the use of the irregular Z-buffer for rendering hard shadows with an algorithm we refer to as *irregular shadow mapping* (See Figure 3). The basic operations performed by this algorithm inform the design of the architecture presented in the next section.

Irregular shadow mapping utilizes the irregular Z-buffer to eliminate the visual artifacts commonly associated with shadow mapping. While conventional shadow mapping can render shadows in complex scenes with high performance, these sampling-related artifacts are sufficiently serious to limit its usefulness in real applications. The algorithm renders the scene first from the position of the light and then from the position of the eye. The contents of the two resulting depth buffers provide the basis for determining which samples in the eye-view are visible from the light (and are thus not in shadow). Unfortunately, this calculation is highly errorprone due to a mismatch between the eye-view and light-view sampling patterns. Irregular shadow mapping overcomes this problem by basing the light-view sampling pattern on the positions of pixels in the eye-view raster and their corresponding depth values.

In the next two subsections we review several key details of the conventional and irregular shadow mapping algorithms. Irregular shadow mapping was previously introduced and compared with existing shadow rendering techniques by Johnson et al. [2004] and Aila and Laine [2004].

Fig. 3. An image from our hardware simulator. The shadows result from irregular shadow mapping via the irregular Z-buffer.



(a) Conventional shadow mapping (b) Irregular shadow mapping

Fig. 4. Conventional versus irregular shadow mapping. In conventional shadow mapping, both the eye-view and light-view images are rendered with the classical Z-buffer, leading to a mismatch between the desired and actual sample locations in the shadow map. Irregular shadow mapping avoids this mismatch by rendering the light-view image with the irregular Z-buffer.

3.1 Conventional Shadow Mapping

Figure 4(a) illustrates conventional shadow mapping and the cause of its artifacts. The scene is rendered first from the light position (yielding z_{near} values) and then from the eye position. Each pixel in the eye view is treated as a 3-space point positioned according to its location in the image plane and its z value

	Conventional Shadow Mapping	Irregular Shadow Mapping	
* 0	render light-view to regular Z-buffer (yields z_{near})		
1	render eye-view to regular Z-buffer (gives points P)	render eye-view to regular Z-buffer - depth values only (gives points P)	(a)
* 2	transform P into light-space (gives P'_z) and project into light-view image plane (P'_{uv})	transform P into light-space (gives P'_z) and project into light-view image plane (P'_{uv})	(b)
* 3		insert P'_{uv} and P'_z into 2D spatial acceleration structure	
* 4		render light-view to irregular Z-buffer locations (P'_{uv}) given by accelerator (yields z_{near})	(d)
5	render eye-view to regular Z-buffer	render eye-view to regular Z-buffer	(e)
* 6	select the light-view sample nearest P'_{uv} and shadow pixel if $z_{near} < z_{sample}$	shadow pixel if $z_{near} < P'_z$	

Fig. 5. Conventional and irregular shadow mapping compared. Steps marked "*" are repeated per light source. Steps (a), (b), (d), and (e) match those shown in the hardware-centric view of irregular shadow mapping seen in Figure 6.

(from the depth buffer). This point is transformed into light space, yielding P' and its distance from the light-view image plane (P'_z). The original eye-space pixel is considered to be in shadow iff $z_{near} < P'_z$, using an estimated z_{near} value. The algorithm estimates z_{near} from the z_{near} values of one or more light-view sample(s) nearest the projection of point P'. This estimation step is the primary cause of artifacts produced by the technique as the estimation error is generally unbounded.

There are several variants of shadow mapping which reduce but do not eliminate sampling and self-shadowing artifacts [Sen et al. 2003; Fernando et al. 2001; Stamminger and Drettakis 2002]. These algorithms do not resolve the fundamental mismatch in sampling patterns between the eye- and light-views, which is the root cause of shadow mapping artifacts.

3.2 Irregular Shadow Mapping

Figure 4(b) illustrates irregular shadow mapping and how it eliminates most self-shadowing and all aliasing artifacts. Pseudocode for the algorithm and its behavior relative to conventional shadow mapping is shown in Figure 5. The scene is first rendered from the eye point. As in conventional shadow mapping, pixels are transformed into light space, yielding P' and P'_z. Unlike conventional shadow mapping, scene geometry is *then* rasterized to the sample positions in the light-view image plane *given by*

the projection of points P', yielding z_{near}. As before, a pixel is in shadow iff $z_{near} < P'_z$. Samples are thereby computed in the light-view image plane precisely where required by pixels in the eye-view plane, and no unnecessary samples are computed.

The density of sample points varies significantly across the image plane (seen in Figure 16(b)), demonstrating the importance of the irregular Z-buffer in this context. Any regular sampling pattern will either undersample some areas and result in artifacts, or oversample other areas, resulting in wasted computation.

4. ARCHITECTURE

The development of the irregular Z-buffer is motivated by the real-time applications described in the previous section. However, the algorithm cannot be implemented efficiently on current graphics hardware, and it does not run at real-time rates on modern CPUs [Aila and Laine 2004]. A new architecture, resulting from evolutionary changes to current GPU designs, is needed. Broadly speaking, the most significant change is to increase the flexibility with which data can be written to memory, but other changes are also necessary. Here we present our modified architecture, describe how it provides efficient support for the irregular Z-buffer algorithm, and briefly consider the cost of its novel features.

4.1 Overview

The irregular Z-buffer requires two distinct phases of computation. These phases are different from those used in a conventional Z-buffer pipeline. The first phase (*construction*) creates the data structure that holds the sample locations. In the second phase (*irregular rasterization*), scene triangles are rasterized to the sample locations encoded in the data structure. Figure 6 shows how these phases ((b) and (d)) are used together in the context of irregular shadow mapping.

Figure 7(a) illustrates our architecture at a high level. It provides efficient support for the construction and rasterization phases of the irregular Z-buffer. The architecture is similar to that of modern GPUs such as the GeForce 6800 [Kilgariff and Fernando 2005] and Eldridge et al.'s [2000] sort-everywhere design. Our GPU includes 16 fragment pipelines, 16 raster operation units, and a 512-bit wide memory interface. The memory system includes four controllers each with two memory chips. The rasterizer is fed by a *geometry processor* (not shown in Figure 7). This processor can access data for all vertices of a given triangle. Though not present in current GPUs, geometry processors are expected to appear in near-future designs.

Our design incorporates several new features. Two of the most important are the abilities to write to computed framebuffer addresses, and to generate an arbitrary number of outputs per input to the fragment processor. The following subsections describe the novel components of our architecture in detail and how their capabilities enable efficient support for the two main phases of the irregular Z-buffer algorithm.

4.2 Fragment Processor

Figure 7(b) shows the design of our MIMD fragment processor. It is composed of a multithreaded core and an L1 texture cache. At this level it resembles a modern fragment processor, except that ours is a true MIMD design rather than a SIMD implementation of a MIMD instruction set [Kilgariff and Fernando 2005]. It utilizes the `NV_fragment_program2` instruction set [Brown and Werness 2004] which includes a conditional branch. We augment this ISA with instructions for logical shifts.

Our fragment processor has three new and potentially challenging to implement capabilities (discussed in detail in Section 6). These include the ability to generate an arbitrary number of output fragments from each input fragment, the ability to specify the framebuffer address to be written for each output fragment (subject to certain constraints), and the ability to perform minimum and maximum

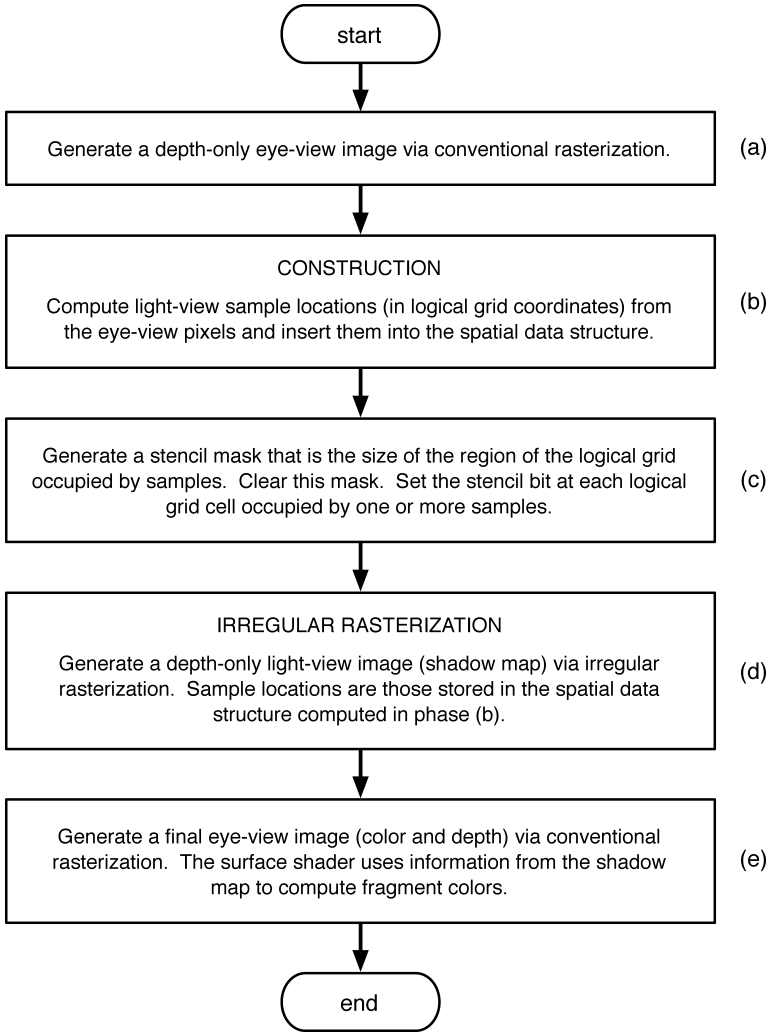
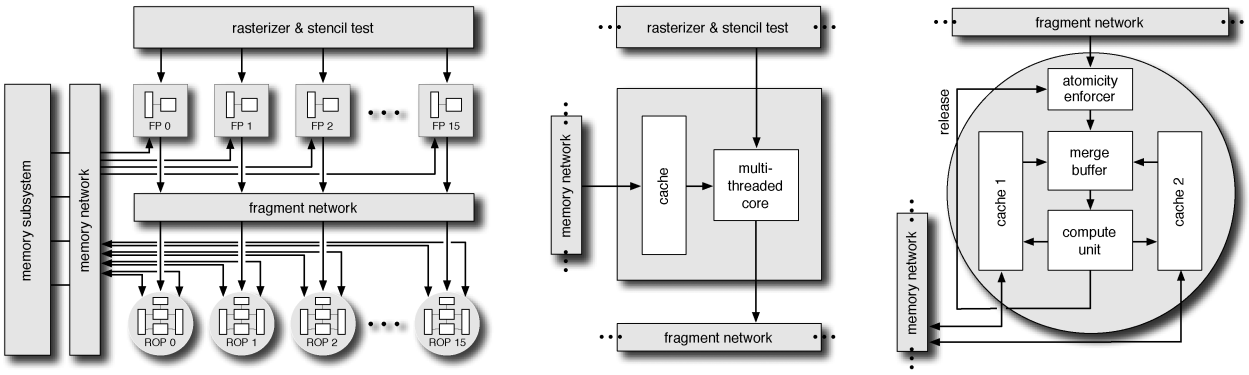


Fig. 6. A hardware-centric view of the irregular Z-buffer as it is used for irregular shadow mapping. Steps (a), (b), (d), and (e) match those in the algorithm-centric view seen in Figure 5. Phases (b) and (d) are shown in greater detail in Figures 8 and 10.

associative reduction operations. Additionally, the fragment processor has access to certain per-triangle data computed by the geometry processor. This data includes the homogeneous equations describing the edges and Z interpolation [Olano and Greer 1997] of the fragment's triangle.

Figure 8 (middle box) illustrates the role of the fragment processor during the construction phase of irregular shadow mapping. The operation of the fragment processor in this phase is straightforward. Of the new capabilities only the reduction operations are necessary. A simple fragment program computes the light-view sample location P'_{uv} for each eye-view pixel P_{ij} and outputs a new node for later insertion into the appropriate linked list by the raster operation unit (ROP). The `min()` and `max()` reduction operations are used to track the bounding box of samples in the light-view image plane. These bounds are used during rasterization for viewport clipping and as the extents of a stencil mask for early rejection of unnecessary fragments.



(a) Overall architecture (b) Fragment processor detail (c) Raster operation unit detail

Fig. 7. The architecture of our machine. It is composed of 16 fragment processors, 16 raster operation units, and four memory controllers each with two DRAM chips. The data paths between the system components shown in (a) are all 16 bytes wide.

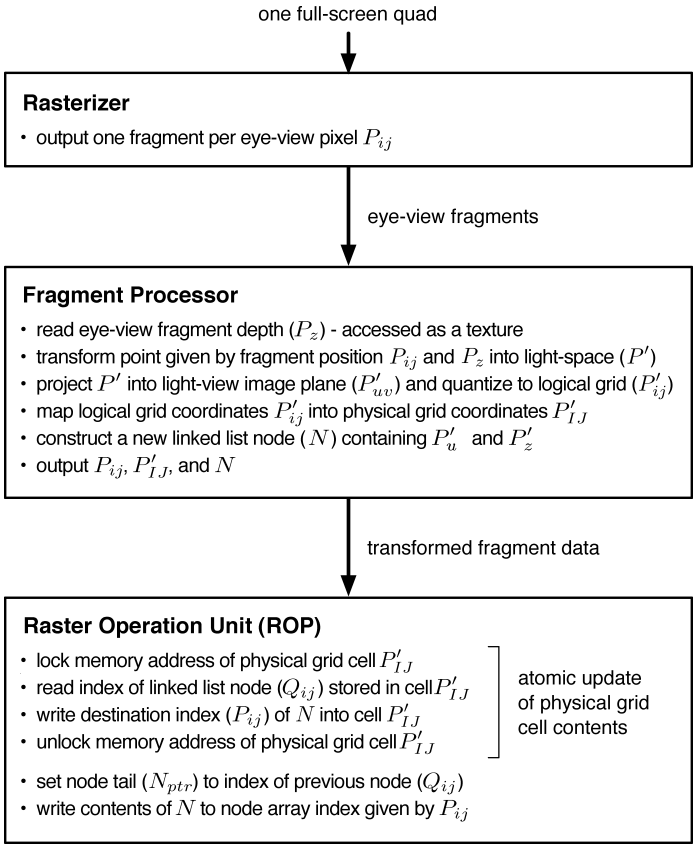
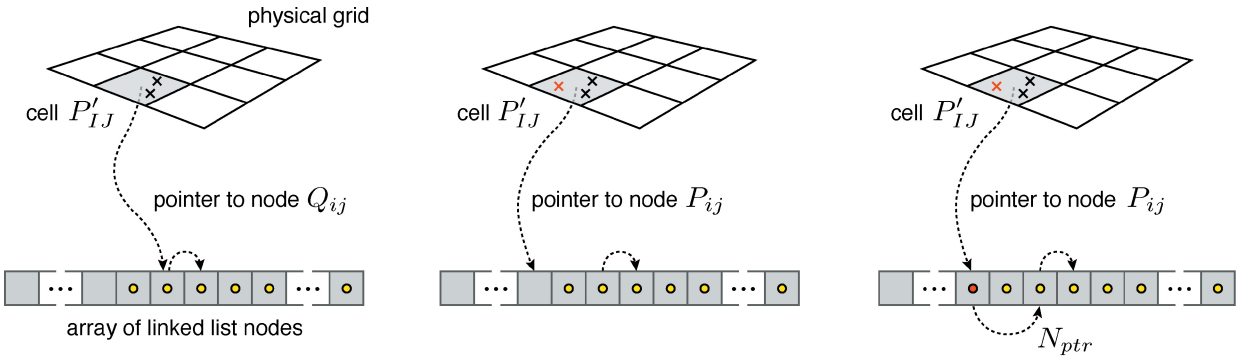


Fig. 8. The construction phase of the irregular Z-buffer algorithm as it is used for irregular shadow mapping on our architecture. Samples are inserted into a grid of linked lists for later use during rasterization. Linked list nodes are stored together in a large array indexed by the position (P_{ij}) of the respective pixel in the eye-view raster.



(a) Data structure before insertion (b) Atomic update of linked list head (c) Node data is stored and tail pointer set

Fig. 9. The steps used to insert a new sample point (denoted by the red "×") into the spatial data structure during the construction phase of the irregular Z-buffer algorithm. All linked list nodes are stored together in a single array. For clarity, only the links between nodes in the list affected by this insertion are shown.

Figure 10 (third box) illustrates the role of the fragment processor in the rasterization phase of irregular shadow mapping. It is here that the ability to generate multiple outputs per input and specify the destination address of each output is needed. The basic operation involves generating one output for each linked list sample covered by a given input fragment. Specifically, the process works as follows. The fixed-function rasterizer generates fragments (hereafter *grid fragments*) as normal, with the one exception illustrated in Figure 11. The fragment processor traverses the linked list of samples in the grid cell associated with each fragment. Samples are tested against the edge equations of the triangle, [Pineda 1988; Olano and Greer 1997]. If the sample is inside the triangle, the fragment processor evaluates the Z interpolation equation at the sample location and generates an output *framebuffer fragment*. The output contains the address of the eye-view pixel corresponding to the sample and the computed Z value, for later use by the ROP.

4.3 Raster Operation Unit

Figure 7(c) shows the details of our raster operation unit (ROP) design. The ROP performs atomic read-modify-write operations to memory. The ROP is a configurable unit in which multiple read-modify-write transactions can be in progress simultaneously. The central challenge in ROP design is to preserve the atomic semantics of the read-modify-write operation. A key insight used in modern GPUs is that atomicity need only be enforced per pixel (i.e., per memory location). Thus, transactions to independent pixels may be arbitrarily intermingled.

Our ROP unit is more flexible than that in current GPUs. In particular, the ROP may generate writes to memory addresses *other* than the fragment's original framebuffer address. For this reason, our ROP uses a pixel cache architecture [Goris et al. 1987] rather than a coalesce buffer [Triantos 2005]. If desired, this second write address can be computed within the ROP from data read from the original framebuffer address.

The ability to generate writes to addresses other than that of the incoming fragment is crucial to the construction phase of the irregular Z-buffer algorithm. The role of the ROP unit in this phase, in the context of shadow mapping, is shown in Figure 8 (bottom box). Each sample node is inserted into the spatial data structure by prepending it onto the linked list of the physical grid cell (P'_{IJ}) given by the fragment processor. The insertion process is illustrated in Figure 9, and occurs in two steps. First,

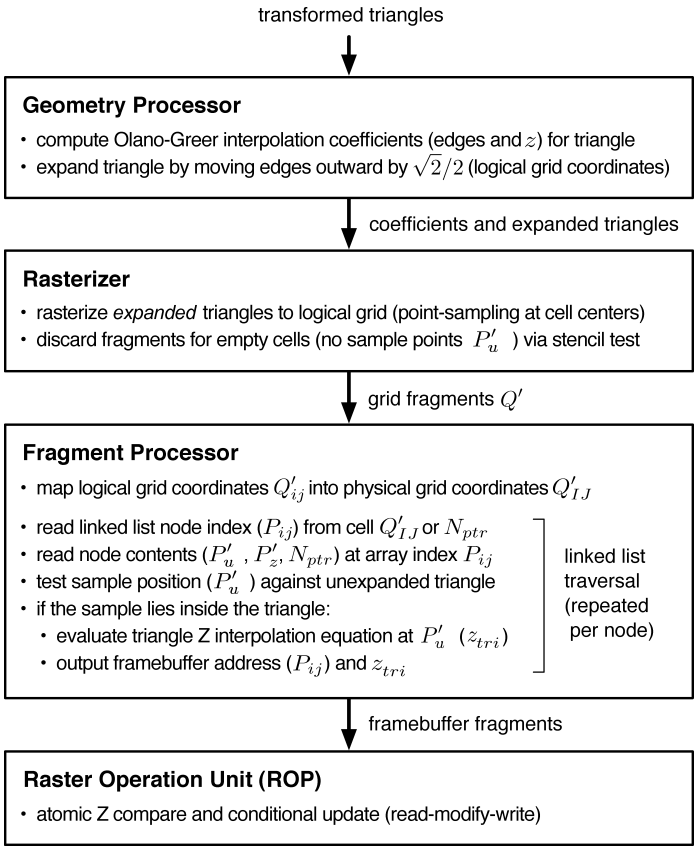


Fig. 10. The rasterization phase of the irregular Z-buffer algorithm as it is used for irregular shadow mapping on our architecture. Triangles are rasterized as normal. The fragment processor queries the spatial data structure for each (grid) fragment to determine which samples fall within the triangle. Triangle expansion (first box) is explained in Figure 11.

the pointer to the head of the linked list (the contents of cell P'_{IJ}) is atomically updated to the index of the new node. Later, the tail of the new node is set to the previous head of the linked list and the node contents are written to memory. This write is to an address included in the data sent by the fragment processor *and is not the framebuffer address of the original fragment*.

In contrast, the operation of the ROP during the rasterization phase of irregular shadow mapping (Figure 10, bottom box) is very similar to that in current GPUs. The ability to write to an arbitrary address is unnecessary. Here, the ROP performs a traditional atomic compare / conditional update of a value stored in the depth buffer. The ROP compares the Z value of the incoming fragment with the value of the respective light-view sample stored in the depth buffer. The stored value is updated depending on the outcome of the comparison.

The ROP operation in both phases is atomic. The *atomicity enforcer* seen in Figure 7(c) ensures the ROP has no more than one transaction in progress for any particular pixel. This unit is similar to that described by VanDyke et al. [2004]. It maintains a table indexed by a hash of the pixel address, indicating whether or not a fragment is in flight for a given hash. If a fragment is in flight, further fragments for that pixel are stalled (in order) in a queue until the offending fragment clears. Meanwhile, fragments for other pixels continue to pass through the atomicity enforcer. The *release* wire signals the

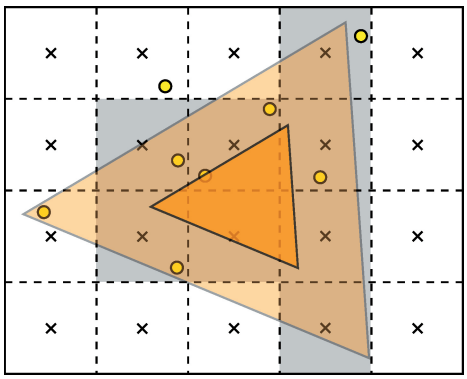


Fig. 11. Triangle edges are moved outward by $\sqrt{2}/2$ prior to scan conversion to enable the use of a conventional rasterizer during irregular rasterization. Classical scan conversion (shaded cells) of an expanded triangle at cell centers ensures (grid) fragments will be generated for any *cell touched by the unexpanded triangle. This is necessary since irregular Z-buffer samples (circles) may lie anywhere within the cell bounds.*

atomicity enforcer when all memory transactions related to a given pixel have completed, at which point it is safe to begin processing another fragment for this pixel. Atomicity is discussed further in Section 6.2.

4.4 Fragment Network

To improve the spatial and temporal reuse of cached framebuffer data, pixel addresses are statically mapped to ROPs. Fragments are routed to the ROP that "owns" the address to which they are being written. The *fragment network* performs this task. It is an $m \times n$ network with internal buffering capable of accepting input from up to m fragment processors and routing output to up to n ROP units each cycle.

In classical Z-buffer architectures, a network of this type is unnecessary as the fragment routing is straightforward. Pixel addresses are partitioned in memory and in the rasterizer such that fragment processors and ROPs are connected one-to-one as are ROPs and memory controllers. In the irregular Z-buffer, this routing scheme is unworkable. For example, consider the construction phase of irregular shadow mapping. The relationship between the positions of pixels in the eye-view raster and the physical grid cells to which the corresponding light-view samples map is scene dependent and thus cannot be known a priori. As a result, a static partitioning of physical grid cell addresses relative to eye-view pixel addresses cannot be set in the rasterizer until the construction phase is ended!

4.5 Cost of Additional Features

It is difficult to estimate the incremental cost of the novel features of our proposed architecture. We have not modeled our design at the RTL level, and there is insufficient public information regarding current GPUs to permit an accurate comparison. However, we can identify several likely sources of additional cost. Most significantly, our MIMD fragment processor requires a small *per-core* instruction cache and decode unit—unnecessary in SIMD designs. Moreover, our $m \times n$ fragment network is more complex than the fast-path routing used currently and thus requires greater die area. Last, the multiport pixel cache design of our ROP necessitates additional cache wiring and control logic.

5. PERFORMANCE

Any new technique targeted at real-time graphics applications must run fast as well as produce images of the desired quality. However, evaluating the performance of new techniques like ours is challenging.

Typically, the performance of an algorithm is measured in one of two ways: at a high level by counting arithmetic operations used for some data set, or at a low level by implementing the algorithm on current hardware and timing its execution. In the case of the irregular Z-buffer, neither approach is adequate. Memory hierarchy effects such as cache misses and associated latencies have become too important for simple operation-counting to yield accurate performance estimates for memory-intensive techniques. However, current architectures do not support our technique well, so timing an implementation of it on these architectures would not be very informative.

Instead, we have constructed a high-level performance simulator for the architecture described in the previous section. It is similar in spirit to the C-model performance simulators used in industry, although it is not as detailed or as broad in scope. We focus on the key performance aspects of the architecture, specifically memory hierarchy and parallelism effects. We model the performance of the memory system in greatest detail, the processor performance at a medium level of detail, and the fixed-function units such as the rasterizer at a functionality level only. In the following subsections we describe our simulator in detail and analyze the performance of the irregular Z-buffer in the context of shadow rendering.

5.1 Simulator

Our simulation environment is composed of two parts, a hardware simulator, and a *supervisory program* which models the functionality of the surrounding system. The supervisory program builds a memory image for the GPU memory system and invokes the simulator with the appropriate parameters for each phase of the irregular Z-buffer algorithm. Following simulation termination, the supervisory program retrieves the depth or shadow map values (depending on the application) from the output memory image of the simulated framebuffer and uses them in the computation of a final image. This step is necessary as our hardware simulator is not configured to perform classical Z-buffer rendering. Figure 3 was generated using this system, with irregular shadow mapping.

Our hardware simulator is built using the Liberty Simulation Environment (LSE) [Vachharajani et al. 2004]. LSE simplifies the development of modular, event-driven, cycle-accurate hardware simulators. It is composed of a structural specification language (and compiler), and a behavioral specification language (C with extensions). The former is used to specify the number, type, and interconnectivity of machine components, and the latter the functionality of each of these components.

At a high level, our LSE machine specification matches the design shown in Figure 7. There is a one-to-one correspondence between the custom LSE modules in our simulator, and the functional units composing the fragment processor, ROP, and fragment and memory networks. At the functional level, the fragment processor is multithreaded (16-way) with a zero-cycle context switch, round-robin scheduling, and a single execution pipeline. It is programmable, and one scalar or 4-wide vector operation can be issued each cycle. We assume all instructions except memory loads complete in one cycle. No effort is made to model stalls due to arithmetic unit latency. The ROP unit is pipelined. Each cycle it can issue up to two loads from the merge buffer and two stores from the compute unit, and retire a single fragment. Multiple fragments can be in-flight at the same time, subject to the atomicity restrictions discussed in Section 4.3. As in conventional designs, this ROP is configurable but not programmable. The operation type (e.g., Z-compare, linked list insertion) is set via a *mode* wire.

Our memory system is based on several modules from the Spinach system (an LSE simulator for network interface controllers) [Willmann et al. 2004], and the DSIM DRAM library [Rixner 2004]. We use the GDDR3 DRAM module from this library. The model honors all bank and channel timing restrictions specified by the manufacturer's data sheet [Micron Technology, Inc. 2003].

Performance Summary

Scene	Lights	Total Samples	Total Cycles	Frame Rate
Doom3	2	2,621,440	44,628,870	11.20 fps
T-Rex	2	2,621,440	43,892,404	11.39 fps

Fig. 12. A summary of the performance of irregular shadow mapping on our hardware simulator. One scene is from Doom3 by Id Software. The other contains a skeleton which casts numerous fine shadow details. All output is 1280 × 1024 pixels in resolution. The totals include the cost of the construction and rasterization phases only (Figures 6(b) and 6(d)). The cost of the initial (a) and final (e) Z-buffer passes and stencil buffer setup (c) are not included. These costs are understood and are not modeled by our simulator. Detailed results are shown in Figure 13.

Construction — Avg Utilization / Stalls — Hit Rate / Avg Miss Latency

Scene, Light	Time (cycles)	Fragment Processors	ROP Units	Cycles / Sample	Frag Proc Cache	ROP Cache 1	ROP Cache 2	Mem Bandwidth Utilization
Doom3, 0	3.0 M	35.8% / 2.4%	2.7% / < 0.01%	2.4	93.8% / 211	96.7% / 372	45.1% / 439	27.8%
Doom3, 1	2.9 M	38.1% / 0.9%	2.9% / < 0.01%	2.2	93.8% / 110	99.9% / 117	48.7% / 226	26.1%
T-Rex, 0	3.0 M	36.1% / 1.1%	2.7% / < 0.01%	2.3	93.8% / 127	99.0% / 192	45.9% / 256	26.1%
T-Rex, 1	2.8 M	39.6% / 1.7%	3.0% / < 0.01%	2.1	93.8% / 155	99.2% / 234	46.6% / 318	28.3%

Rasterization — Avg Utilization / Stalls — Hit Rate / Avg Miss Latency

Scene, Light	Time (cycles)	Fragment Processors	ROP Units	Cycles / Fragment	Frag Proc Cache	ROP Cache 1	ROP Cache 2	Mem Bandwidth Utilization
Doom3, 0	22.9 M	37.0% / 54.4%	1.4% / < 0.01%	4.3	59.7% / 699	56.7% / 378	57.6% / 375	40.3%
Doom3, 1	15.8 M	36.5% / 42.7%	2.2% / < 0.01%	2.8	63.5% / 766	56.2% / 402	56.6% / 397	38.7%
T-Rex, 0	20.8 M	54.5% / 36.9%	1.0% / < 0.01%	6.3	73.7% / 504	66.9% / 488	64.9% / 488	38.4%
T-Rex, 1	17.3 M	55.1% / 32.3%	1.1% / < 0.01%	5.5	74.0% / 481	64.0% / 435	64.7% / 443	37.9%

Fig. 13. A detailed view of the results summarized in Figure 12. These results illustrate the performance of shadow mapping on our irregular Z-buffer hardware simulator. Columns 3 and 4 express the average time spent by the fragment processors and ROP units on useful work and the average idle time due to memory latency-induced stalls. The average miss latencies reported in columns 6–8 are measured in cycles and result from our use of numerous internal queues which increase throughput at the cost of latency. The last column reports the DRAM bandwidth utilization as a fraction of theoretical maximum bandwidth.

5.2 Results and Analysis

We simulate the construction and rasterization phases of the irregular Z-buffer as it is used for irregular shadow mapping (Figures 6(b) and 6(d), respectively). These results are summarized in Figure 12, and do not include the operations represented in Figure 6 phases (a), (c), and (e). Since these phases do not require or exercise our proposed architectural additions, their performance is already well understood.

Figure 13 (top) details the simulated performance of the irregular Z-buffer during the construction phase of irregular shadow mapping, under the machine configuration detailed in Figure 14. The utilization of the fragment processors is relatively low due to transient load imbalances rather than memory-induced stalls.

Several characteristics of the resulting spatial data structures are summarized in Figure 15 in the columns denoted "Construction." Doom3 light 0 represents a challenging case. The light-view samples are primarily concentrated within a localized region in the image plane. This light sits near one wall which is visible over a large fraction of the eye-view viewport. With the light-view normal nearly parallel to the wall, the points on this wall visible from the eye (points P') project to a narrow band. The high concentration of points should result in reduced utilization of the physical grid and longer linked lists. However, the formation of such hot spots in the data structure is reduced by the presence of the logical grid and the function which maps between it and the physical grid.

System Configuration — Cache Configuration

Clock Rate	Fragment Processors	ROP Units	Memory Controllers	DRAM Chips	Size	Line Size	Block Size	Assoc.	Hit Latency
500 MHz	16	16	4	8	16 KB	16 B	64 B	8	1

Fig. 14. The configuration of our hardware simulator. All units including the memory interfaces are clocked at the rate shown, and all caches share the indicated configuration.

Scene Details — Construction — Rasterization

Scene, Light	Triangle Count	Avg. Depth Complexity	Physical Grid Utilization	Cell List Length	Stencil Buffer Size	Grid Fragments Pre / Post Stencil	Framebuffer Fragments Pre / Post Z-Test
Doom3, 0	7,581	4.1	21.3%	1/23/2195	553 x 501	1.5 M / 0.5 M	5.3 M / 1.0 M
Doom3, 1	7,581	4.3	8.4%	1/59/1002	175 x 202	0.2 M / 0.2 M	5.7 M / 1.2 M
T-Rex, 0	69,840	2.5	25.3%	1/20/821	1346 x 407	0.8 M / 0.3 M	3.3 M / 1.3 M
T-Rex, 1	69,840	3.0	27.3%	1/18/1051	937 x 320	0.3 M / 0.3 M	3.2 M / 0.9 M

Fig. 15. Details for each scene/light combination used in our simulations of irregular shadow mapping. The third column of the table denotes the average depth complexity at a light-view sample. The fourth column reports the percentage of nonempty physical grid cells as a fraction of the total 512^2 cell count. The fifth column shows the (nonempty) minimum, average, and maximum linked list lengths. The stencil buffer sizes reported in the sixth column match the effective logical grid bounds computed via the `min()` and `max()` reduction operations across the fragment processors.

Figure 13 (bottom) details the performance of the irregular Z-buffer during the rasterization phase of irregular shadow mapping under the same machine configuration. Our simulation assumes that the stencil test described in Section 4.2 rejects grid fragments for empty logical grid cells. While we do not model this stencil buffer memory traffic, the additional bandwidth consumed can be computed from the columns in Figure 15 marked "Rasterization." The relatively low utilization of the fragment processors is due to memory-induced stalls and indicates that the performance of the memory system bounds this phase of the algorithm.

Rasterization dominates the execution time of the irregular Z-buffer. For the scenes tested, the rasterization phase required between four and seven times the number of cycles as the construction phase. Although the cost of rasterization depends strongly on the geometric and depth complexity of the scene, our timings indicate that the cost of data structure construction is secondary. This result is interesting as the construction phase is more architecturally challenging to support due to the need for atomic read-modify-write operations during linked list creation.

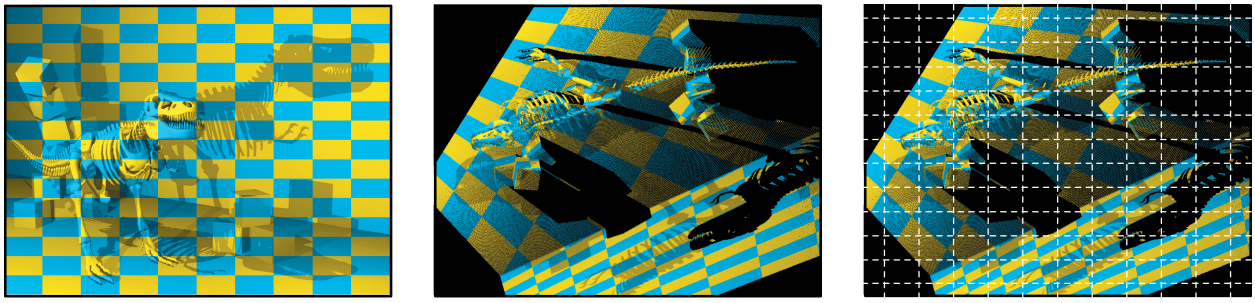
These timings result from a reasonable effort to tune our simulator. Our tuning strategy has focused on applying well-understood optimizations to reduce pressure on key resources (e.g., memory bandwidth) in the system. An in-depth discussion of the role of memory tiling and locality, and additional optimizations which may further improve the performance of our system, are among the topics discussed in the next section.

6. DISCUSSION

In this section we discuss several of the finer points of our architecture and of its interaction with applications such as irregular shadow mapping.

6.1 Locality

Exploiting the locality of memory references is key to the performance of the irregular Z-buffer. At first glance, the goal of achieving high reuse might seem irreconcilable with the unpredictable composition



(a) (b) (c)

Fig. 16. The view of the tyrannosaurus rex scene from Figure 3 is overlaid with colored tiles in (a). A portion of the irregular shadow map for "light 1" is seen in (b). These points are colored by the tile of the respective eye-view pixel. The light-view image plane is overlaid in (c) with a grid representing the bounds of tiles of logical grid cells. While the eye-view and light-view tiles do not align, pixels nearby in the eye-view raster tend to remain nearby following transformation into the light-view logical grid.

and patterns of access of our main data structure. However, we have been able to exploit several sources of spatial (and to a lesser degree temporal) locality during both the construction and rasterization phases of irregular shadow mapping.

As we noted earlier, the relationship between the positions of pixels in the eye-view raster and the corresponding samples in the light-view image plane is scene dependent. However, this relationship is *not* necessarily incoherent. In fact, pixels that are nearby in the eye-view tend to stay nearby after transformation into the light-view. Consider Figure 16. The same view of the tyrannosaurus rex scene from Figure 3 is shown in Figure 16(a) with the raster partitioned into tiles as denoted by the colored overlay. The contents of the irregular Z-buffer are shown in Figure 16(b) for one of the light sources in the scene. The irregular Z-buffer sample points are colored according to the tile containing the respective eye-view pixel. Observe that tiles in the eye-view raster remain largely intact following the transformation and projection of the pixels into the light-view image plane. The degree to which this property holds depends upon the amount of local variation in the per-pixel depth of the geometry directly visible from the eye. The greater the variation, the more the corresponding light-view samples will be dispersed.

Our data structures (eye-view raster, light-view physical grid, and array of linked list nodes) are tiled in memory and we strive to aggregate work related to the same tile on the same functional unit, nearby in time [McCormack et al. 1998]. Although the footprint of a tile of transformed pixels will almost never align with a tile in the logical/physical grid,[1] some amount of locality is clearly preserved. Figure 16(c) illustrates this phenomenon. The spatial locality we exploit during both the construction and rasterization phases of irregular shadow mapping results from this property. Additionally, we are able to exploit temporal reuse in the case where small, adjacent triangles are rasterized in quick succession.

6.2 Atomicity

Recall that a key feature of our ROP unit is its ability to write to addresses computed from data read from the incoming pixel address or from the fragment processor. These indirect writes occur after the pixel address has cleared the atomicity enforcer. As a result the addresses of the writes cannot be known

[1]Our logical-to-physical grid mapping function preserves the integrity of tiles. Tiles in the logical grid map to tiles in the physical grid.

(and thus protected) by the enforcer, potentially leading to race conditions and nondeterminism. It is the responsibility of the software driver and/or the programmer to ensure that this does not occur.

An example of an indirect write occurs during the construction phase of irregular shadow mapping. Here, the ROP issues two writes. The first updates the value (the address of the node at the head of the linked list) stored at the incoming pixel address. The second writes the contents of the new linked list head node. The address of this node is computed from data provided by the fragment processor, and is unknown to the atomicity enforcer. However, a property of this phase of the algorithm is that each node is written only once (each sample is inserted into the data structure once). Moreover, our pixel caches support selective writes—only dirty subregions of a block are written out to memory. As a result, though a given linked list node may exist in multiple cache lines on the same ROP (or in different ROPs), each node is guaranteed to be written to memory only once.

Although avoided in this example, nondeterminism cannot be guaranteed in all cases. By exposing the irregular Z-buffer through a high-level OpenGL/DirectX API instead of as a set of low-level ROP capabilities, the driver can ensure that the machine is never used in an unsafe manner.

6.3 Maintaining Fragment Order

Real-time rendering APIs define precise ordering semantics for noncommutative operations such as blending and Z-buffered color writes. In some cases these semantics are directly useful to application programmers, but they are also important to guard against nondeterminism from frame to frame or from one hardware generation to another.

Fortunately, *order does not matter* when using the irregular Z-buffer to generate shadow or other Z-only maps. In the construction phase of irregular shadow mapping, the ordering of nodes in each linked list is unimportant and opaque to the user if the data structure is hidden behind a high-level API as we've suggested previously. In the rasterization phase, fragment order is not visible to the user because the Z comparison and update operation is commutative so long as it does not carry auxiliary information with it, such as color.

However, the irregular Z-buffer can be used for different applications in which color *is* carried with Z (e.g., reflection map generation). For these applications, the preservation of fragment order during rasterization would matter.

Preserving order in a parallel architecture can be difficult. With current API definitions, architectures can maintain order by processing fragments in SIMD lockstep. In our architecture, this solution is impractical due to the ability of our fragment processors to compute the address of their output fragments and generate a variable number of outputs for each input fragment.

In the general case, preserving fragment order would seem to require an (expensive) global order enforcement mechanism. Order cannot be maintained on a per-pipe basis, because we do not know what the destination address will be when a fragment enters a pipe. However, the irregular Z-buffer does not require the full generality of the functionality just described. In particular, during rasterization, we can guarantee that the fragments generated by a fragment program will always belong to a particular logical grid cell. Thus, if we route fragments to fragment processors based on their logical grid cell (as we do anyway), we simplify the problem to one of enforcing order locally within each fragment pipeline. Although we have not yet implemented the necessary per-pipe order enforcement in our simulator, we are confident that it is possible. Again, to guarantee that programmers cannot implement non-deterministic code, some of the basic hardware functionality must be hidden behind higher-level APIs.

6.4 SIMD Fragment Programs

Hardware that uses a SIMD execution model can be built with lower cost than MIMD hardware, because instruction caches and decoders can be shared by many processing units. Our fragment processor is

```
// Inputs:
//   edge_coeff = interpolant for edges of unexpanded tri
//   z_coeff    = interpolant for 1/Z across triangle face
//   lgrid.xy   = logical grid coords of incoming fragment
//
// Outputs:
//   out.index  = framebuffer location of outgoing fragment
//   out.z      = interpolated 1/Z of outgoing fragment
//
// Threads iterate through the following operations in lock-
// step, each on its own fragment.  A stencil test discards
// fragments for empty logical grid cells in an earlier step.

// initialize "got_work" and "sample_index"
create_thread();

repeat {
  if (!got_work) {                          // predication
    pgrid_index  = hash(lgrid.xy);
    sample_index = pgrid[pgrid_index];      // texture fetch
    got_work = TRUE;
  }
  if (sample_index != NULL) {               // predication
    sample = NodeArray[sample_index];       // texture fetch
    if (inside_tri(edge_coeff, sample.xy)) {  // predication
      out.z = interp_Z(z_coeff, sample.xy);
      out.index = sample_index;
      emit(out);
    }
    sample_index = sample.next;
  }
} until (sample_index == NULL);

destroy_thread();
```

Fig. 17. Fragment processor pseudocode for irregular rasterization. The code is invoked for each grid fragment (refer to Figure 10) and is suitable for SIMD execution. Our fragment processor follows an MIMD execution model. The MIMD code is identical to that shown here, except that threads do not iterate through the loop in lockstep.

currently MIMD. However, both phases of the irregular Z-buffer can be efficiently implemented using a SIMD execution model, as long as the fragment processor supports conditional inputs and conditional outputs like those of the SIMD Imagine machine [Kapasi et al. 2000]. A SIMD implementation of irregular rasterization is illustrated in Figure 17.

6.5 Storage Organization

Our implementation of irregular shadow mapping has the unusual property that it does not explicitly allocate memory even though the algorithm constructs a dynamic data structure. Explicit allocation is unnecessary as storage is implicitly reserved for each linked list node. The reserved memory is associated with the original eye-view pixel that resulted in the shadow map sample contained in the node. This strategy is possible due to the one-to-one correspondence between eye-view pixels and light-view samples.

The advantages of avoiding explicit memory allocation include freedom from potential serialization bottlenecks in the memory allocator, and a reduced chance of unbalanced allocation across memory partitions. However, there is an important disadvantage. Since the storage location of a linked list node is associated with the position of the respective pixel in the eye-view, it is *not* associated with the position of the node's sample in the light-view. As a result the any-to-any routing capability of the fragment network is required for irregular rasterization of the light-view(s) during shadow mapping.

An alternative design explicitly allocates memory for each linked list node. The node is placed in the memory partition associated with the region of the light-view containing the node's sample point. This design eliminates the need for an any-to-any fragment network during irregular rasterization. However, such a network is still required during the construction phase for the reason described in Section 4.4.

6.6 Additional Optimizations

Our simulated architecture does not include some of the more complex optimizations used by current GPUs such as hierarchical early Z-culling [Greene et al. 1993] and framebuffer compression. These optimizations are not well documented in the open literature and they are challenging to implement effectively even for the large architecture teams at companies such as NVIDIA and ATI.

Although we did not implement these optimizations, we have given them some consideration. In particular, we believe that hierarchical Z-culling could be implemented at the granularity of logical grid cells. During irregular rasterization, if a fragment is generated for one or more samples in a cell and the cell is fully covered by the projected triangle, the fragment processor recomputes the depth bounds of the cell. The new value is the minimum of the maximum sample depth and the maximum triangle depth at each of the cell corners. The result is passed on to the hierarchical Z subsystem.

6.7 Additional Applications

The basic irregular Z-buffer algorithm can be used for any application in which the optimal arrangement of sample locations in the image plane is not a regular grid. For example, high-quality adaptive supersampling can be achieved in an efficient manner. An initial (regular) rasterization pass is used to find discontinuities in the image. Irregular rasterization is used in a subsequent pass to generate additional samples in exactly the regions of the image which require them. Moreover, the number of additional samples in any given region can vary based on the degree of the detected discontinuity. Antialiasing techniques based on the classical Z-buffer cannot achieve this level of flexibility or efficiency. For example, the SAGE architecture [Deering and Naegle 2002] supports nonuniform sampling patterns within a pixel, but the per-pixel sample count is fixed.

Additionally, the irregular Z-buffer is potentially useful in the final gather phase of global illumination computations, as an enhancement to the techniques described by Hachisuka [2005] and Lengyel and Walter [1995]. Beyond specific applications, the architectural enhancements presented here enable hardware acceleration of new classes of algorithms. In particular, the lack of scatter capability in current GPUs is a widely acknowledged limitation that is resolved by our design [Buck 2005; Purcell et al. 2002].

7. CONCLUSION

We have presented several results in this article. First, we have extended the Z-buffer algorithm to support arbitrary sample locations in the image plane. Second, we have demonstrated that the irregular Z-buffer can be used to generate shadow-mapped shadows, and that its use eliminates the artifacts traditionally associated with shadow mapping. Last, we have proposed a set of architectural enhancements in support of the irregular Z-buffer, and demonstrated in simulation how the enhanced design enables rendering of high-quality shadows at interactive frame rates.

One insight we have gained from our work is that atomic read-modify-write operations can be handled by a dedicated unit close to the memory which owns the data. In general-purpose parallel architectures, these operations are typically implemented close to the processor, in an L1 or L2 cache with a complex cache coherency mechanism. Moving such operations to specialized units near memory seems broadly promising. Though the idea has been explored in the context of streaming processors [Ahn et al. 2005], it should be considered more fully by the CPU community in future chip-multiprocessor architectures.

This work also has broader implications. A key characteristic of the irregular Z-buffer approach is that it builds and traverses a simple irregular data structure. Irregular data structures are commonly used in offline rendering systems, but can be useful in real-time rendering systems as well. For example, these structures are used for A-buffers, photon maps, and raytracing acceleration structures. Lefohn et al. [2005] and Lefebvre et al. [2004] have explored the construction and traversal of octree-like data structures on current GPUs. However, the efficiency and generality of these solutions are constrained by the capabilities of the available graphics architectures. We believe the approach presented in this article is an initial step toward providing hardware support for the *efficient* creation and traversal of *broad classes* of spatial data structures in near-future GPUs.

ACKNOWLEDGMENTS

Chris Lundberg developed the supervisory program which drives our simulator. Scott Rixner provided access to his DSIM memory controller simulator. Vijay Pai and Paul Willmann provided access to their Spinach architecture simulator. Ikrima Elhassan wrote the initial version of our Doom3 scene parser. Don Fussell provided a variety of useful suggestions, and in particular was the first of several people to urge us to investigate a grid-of-lists data structure as an alternative to the k-d-tree data structure we started with. Kelly Gaither and Jay Boisseau of the Texas Advanced Computing Center strongly encouraged and supported our work. The work environment they created made this research possible. J. Mike O'Connor, Karthikeyan Sankaralingam, Stephen Keckler, and Doug Burger gave us useful advice on architectural choices and simulation.

REFERENCES

AHN, J. H., EREZ, M., AND DALLY, W. J. 2005. Scatter-add in data parallel architectures. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture* (HPCA-11 2005). 132–142.

AILA, T. AND LAINE, S. 2004. Alias-free shadow maps. In *Proceedings of the Eurographics Symposium on Rendering 2004*. Eurographics Association, Aire-la-Ville, Switzerland, 161–166.

APPEL, A. 1968. Some techniques for shading machine renderings of solids. In *Proceedings of the AFIPS Spring Joint Computer Conference*. Vol. 32. 37–45.

BROWN, P. AND WERNESS, E. 2004. GL_NV_fragment_program2 extension specification. Go to the Web site `http://oss.sgi.com/projects/ogle-sample/registry/NV/fragment-program2.txt`.

BUCK, I. 2005. Taking the plunge into GPU computing. In *GPU Gems 2: Programming Techniques for High Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, Reading, MA, 509–519.

CATMULL, E. 1974. A subdivision algorithm for computer display of curved surfaces. Ph.D. dissertation. Department of Computer Science, University of Utah, Salt Lake City, UT.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The reyes image rendering architecture. In *SIGGRAPH '87: Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. ACM Press, New York, NY, 95–102.

DEERING, M. AND NAEGLE, D. 2002. The SAGE graphics architecture. In *SIGGRAPH 2002: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. ACM Press, New York, NY, 683–692.

ELDRIDGE, M., IGEHY, H., AND HANRAHAN, P. 2000. Pomegranate: a fully scalable graphics architecture. In *SIGGRAPH '00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. ACM Press, New York, NY/Addison-Wesley Reading, MA, 443–454.

FERNANDO, R., FERNANDEZ, S., BALA, K., AND GREENBERG, D. P. 2001. Adaptive shadow maps. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (ACM SIGGRAPH 2001). ACM Press, New York, NY, 387–390.

GORIS, A., FREDRICKSON, B., AND HAROLD L. BAEVERSTAD, J. 1987. A configurable pixel cache for fast image generation. *IEEE Comput. Graph. Applic.* 7, 3 (Mar.), 24–32.

GREENE, N., KASS, M., AND MILLER, G. 1993. Hierarchical Z-buffer visibility. In *SIGGRAPH '93: Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. ACM Press, New York, NY, 231–238.

HACHISUKA, T. 2005. *GPU Gems 2: Programming Techniques for High Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, Chapter High-Quality Global Illumination Rendering Using Rasterization, 615–633.

JOHNSON, G. S., MARK, W. R., AND BURNS, C. A. 2004. The irregular Z-buffer and its application to shadow mapping. Tech. rep. TR-04-09. Department of Computer Sciences, The University of Texas at Austin, Austin, TX.

KAPASI, U. J., DALLY, W. J., RIXNER, S., MATTSON, P. R., OWENS, J. D., AND KHAILANY, B. 2000. Efficient conditional operations for data-parallel architectures. In *MICRO 33: Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*. ACM Press, New York, NY, 159–170.

KILGARIFF, E. AND FERNANDO, R. 2005. The GeForce 6 series GPU architecture. In *GPU Gems 2: Programming Techniques for High Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, Reading, MA, 471–491.

LEFEBVRE, S., HORNUS, S., AND NEYRET, F. 2004. All-purpose texture sprites. Tech. rep. 5209. INRIA, Rocquencourt, France.

LEFOHN, A. E., KNISS, J., STRZODKA, R., SENGUPTA, S., AND OWENS, J. D. 2005. Glift: Generic, efficient, random-access GPU data structures. *ACM Trans. Graph. 24*. To appear.

LENGYEL, J. AND WALTER, B. 1995. The path-buffer. Tech. rep. PCG-95-4. Program of Computer Graphics, Cornell University, Ithaca, NY.

LINDHOLM, E., KILGARD, M., AND MORETON, H. 2001. A user-programmable vertex engine. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques* (ACM SIGGRAPH 2001). ACM Press, New York, NY, 149–158.

MCCORMACK, J., MCNAMARA, R., GIANOS, C., SEILER, L., JOUPPI, N. P., AND CORRELL, K. 1998. Neon: A single-chip 3D workstation graphics accelerator. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*. ACM Press, New York, NY, 123–132.

MICRON TECHNOLOGY, INC. 2003. Micron Technology 256 Mb (x32) GDDR3 SDRAM datasheet. Micron Technology, Inc., Boise, ID.

OLANO, M. AND GREER, T. 1997. Triangle scan conversion using 2D homogeneous coordinates. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*. ACM Press, New York, NY, 89–95.

PINEDA, J. 1988. A parallel algorithm for polygon rasterization. *Comput. Graph. (SIGGRAPH '88 Proceedings) 22*, 4 (Aug.), 17–20.

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. In *SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. ACM Press, New York, NY, 703–712.

PURCELL, T. J., DONNER, C., CAMMARANO, M., JENSEN, H. W., AND HANRAHAN, P. 2003. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. Eurographics Association, Aire-la-Ville, Switzerland, 41–50.

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, Aire-la-Ville, Switzerland, 299–306.

RIXNER, S. 2004. Memory controller optimizations for web servers. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*. 355–366.

SEN, P., CAMMARANO, M., AND HANRAHAN, P. 2003. Shadow silhouette maps. *ACM Trans. Graph. 22*, 3, 521–526.

STAMMINGER, M. AND DRETTAKIS, G. 2002. Perspective shadow maps. In *SIGGRAPH '02: Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*. ACM Press, New York, NY, 557–562.

TABELLION, E. AND LAMORLETTE, A. 2004. An approximate global illumination system for computer generated films. *ACM Trans. Graph. 23*, 3, 469–476.

TRIANTOS, N. 2005. Windowing system on a 3D pipeline. White paper. NVIDIA Corporation, Santa Clara, CA.

VACHHARAJANI, M., VACHHARAJANI, N., PENRY, D. A., BLOME, J., AND AUGUST, D. I. 2004. The liberty simulation environment, version 1.0. *Perform. Eval. Rev. (Special Issue on Tools for Architecture Research) 31*, 4 (Mar.).

VAN DYKE, J. M., VOORHIES, D. A., JAMES E. MARGESON, I., AND MONTRYM, J. 2004. System, method and article of manufacture for an interlock module in a computer graphics processing pipeline. Patent #6734861 NVIDIA Corporation, Santa Clara, CA.

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

WILLMANN, P., BROGIOLI, M., AND PAI, V. S. 2004. Spinach: A Liberty-based simulator for programmable network interface architectures. In *LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools*. ACM Press, New York, NY, 20–29.

Received June 2005; accepted August 2005