Dynamic 3D Graphics Workload Characterization and the Architectural Implications

Tulika Mitra   Tzi-cker Chiueh

Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400

{mitra, chiueh}@cs.sunysb.edu

Abstract

Although PC-class 3D graphics hardware has made significant strides in the last several years, the underlying architectural design principles are still generally considered as a black art. The quantitative approach prevalent in mainstream computer architecture design is rarely applied, at least as far as publicly available research literature is concerned. One main reason for this deficiency is the absence of a detailed workload characterization of 3D applications. This paper reports the results of a *dynamic* 3D workload characterization effort, which, to the best of our knowledge, is one of the first such studies that have ever been attempted. This study is different from previous similar studies because it focuses on *dynamic* behaviors of 3D applications, specifically, correlations of workload statistics among neighboring frames in interactive 3D applications. Such *inter-frame coherence* is the basis of many performance optimization techniques, but has never been actually quantified in a systematic way. In addition, whenever possible this paper presents the empirical workload characteristics in the light of their architectural implications and detailed design tradeoffs. In particular, the paper discusses quantitatively the extent to which inter-frame coherence impacts novel graphics architecture features such as texture memory management, speculative rendering using predicted viewpoints/viewangles, and dynamic tiling.

1 Introduction

As desktop machines evolve from platforms for business data/document processing to interactive multimedia data access and communications engines, the design focus of the underlying hardware architecture is gradually driven more by requirements from these newer generations of applications. An important class of media-rich applications are critically dependent on high-performance 3D graphics technology, whose ultimate aim is to render photo-realistic images at a video-like frame rate. Accordingly, a large amount of industrial efforts have been spent in recent years on 3D graphics hardware development. From the fierce competition among the PC-class 3D graphics chip vendors, and the amazing technological advances made by them, it is predicted [Kir98] that by Year 2001, the performance of these graphics chips will surpass the designs from the current high-end graphics machine market leader, Silicon Graphics.

Despite the impressive strides made by 3D graphics hardware technology in the last three years [Mac96], the theory and practice of 3D graphics architecture are still relatively under-developed, mainly due to the following two reasons. First, the micro-architecture of almost all commercial 3D graphics chips is not documented in any public literature. Most of the available information contains either only general block diagrams of the chips or performance "claims" that do not shed any lights on their design details. Second, most of the earlier research on 3D graphics hardware tends to be less quantitative than qualitative. In particular, graphics hardware papers that discuss architectural tradeoffs based on concrete measurements from real-world applications only start to appear in the last two years [HG97, CB97, CBS98, C^{+}98]. As a result, it has been impossible to empirically compare various graphics architectural ideas on an unbiased basis, and thus advance the field by drawing on lessons distilled from the comparisons. While it is difficult to solve the first problem, we believe the graphics hardware research community can easily address the second by emulating the experimental research methodology as practiced by the CPU architecture research community for years: reporting detailed quantitative statistics from either prototype measurements or trace-driven simulations [HP96, Raf77].

The goal of this work is to perform a detailed quantitative workload characterization of an important class of 3D graphics applications that could be used to guide the evolution of future 3D graphics processor architecture. Specifically, we focus on *dynamic* polygon-based 3D graphics workloads. By polygonal graphics, we mean the basic primitives used in 3D graphics applications are polygons, which are supported by all existing 3D chips, rather than other geometric primitives such as volumes or splines. By dynamic graphics, we mean 3D workloads that permit the study of rendering dynamics across image frames rather than within individual images. This study is thus quite different from and much more wide in scope than the study reported in [CL97].

Most published work related to 3D graphics workload characterization focused only on specific statistics. Whelan's thesis [Whe85] characterized the distribution of objects on the screen to help decide the partitioning strategy used in multiprocessor rendering. He used six images for the characterization study, most of whose complexity is relatively small by today's standard. Dunwoody [DL90] instrumented a SGI rendering system to collect a trace of graphics library calls, from which a profiler program can compute workload statistics and a performance measurement tool can estimate the running time on a given hardware platform. The focus of that work was mainly on the design and implementation of the tracing environment, rather than the detailed analysis of 3D workloads themselves. Cox and Hanrahan [CH92] discussed a particular aspect of 3D graphics workload, depth complexity, and its implications on parallel polygonal rendering. Perhaps the most comprehensive 3D graphics workload study so far is Ricki Blau's Ph.D. thesis [Bla92]. She studied nine highly complicated images from different periods of Pixar's animation productions by instrumenting proprietary rendering tools. Most of her work is also focused on depth complexity and visible surface measurements. Compared with previous works, the set of statistics collected for our work is larger in scope, and the analysis presented in this paper is specifically oriented towards architectural tradeoffs in graphics hardware design.

The rest of the paper is organized as follows. In Section 2, we briefly review the basic polygonal 3D graphics pipeline. In Section 3, we describe the benchmark applications and the experiment methodology. Section 4 provides detailed characterization of the benchmarks, and their architectural implication from system designer's point of view. In Section 5, we summarize the major findings of this work and provide an outline of future work.

2 Basic Polygonal 3D Graphics Pipeline

3D polygonal rendering starts by traversing the set of geometric primitives that constitute a 3D model. For 3D graphics rendering hardware, each primitive usually consists of easily manipulable units, such as triangles. The rest of the rendering process consists of two distinct stages [FvF+90]: *geometric transformation* and *rasterization*. The *geometric transformation* stage maps the triangles from some 3D coordinate system to 2D coordinates by performing a series of transformations. The computation in this stage is mostly floating-point intensive, involving linear algebraic operations such as matrix multiplication and inner products. Today, in PC-based systems, *geometric transformation* is mostly performed by the general purpose processor, with adequate floating point computation power. The *rasterization* stage converts the transformed triangles into pixel values and stores them in frame buffer for display. This stage involves mostly integer arithmetic, such as simple additions and compares, and requires special purpose processors to achieve reasonable performance. The majority of 3D graphics cards accelerate the *rasterization* stage. In this paper, we will concentrate on workload characterization for the *rasterization* stage of the rendering system.

The *rasterization* stage itself is comprised of two steps. The *scan conversion* step decomposes a triangle into a set of pixels, and calculates the attributes, such as color, depth, alpha, and texture coordinates for each pixel. The *pixel processing* step performs *texture mapping*, *depth test*, and *alpha blending*, for individual pixels.

There are two distinct mechanisms that are quite popular for the *scan conversion* step: *linear interpolation* algorithm and *linear edge function* algorithm. In *linear interpolation* based algorithms [FvF+90], the *triangle set-up* step first computes the slopes, with respect to X axis, for all the attributes, along each edge of the triangle. Next, the *edge processing* step iterates along the edges,

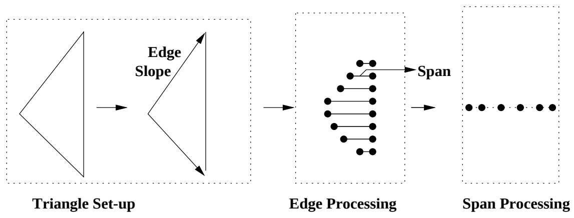


Figure 1: *Scan Conversion of a Triangle using Linear Interpolation algorithm.*

and computes the two end points of a horizontal pixel segment, called *span*. Finally, the *span processing* step iterates along the span, and computes the attributes for each pixel on the span, through linear interpolation (Figure 1).

In *linear edge function* based algorithms [F^+85], each edge of the triangle is defined by a linear edge function. The triangle is scan converted by evaluating, at each pixel's center, the function for all edges, and processing only those pixels that are inside all the edges. The attributes are also computed from the linear functions. Usually, the triangle is traversed by proceeding down from a starting point, and moving outward from the center line [Pin88]. The center line shifts to the left or right, if it steps outside of the triangle at any point of time (Figure 2A). To achieve parallelism, the triangle may be traversed one *pixelstamp* at a time, instead of a single pixel [Pin88]. The *pixelstamp* is an array of pixels of dimension $X \times Y$. The evaluation of edge functions for all the pixels within a pixelstamp is done in parallel, and only qualified pixels are sent to the pixel processing stage. Triangle traversal visits all pixelstamps that are completely or partially inside the triangle (Figure 2B).

The *pixel processing* stage can perform *texture mapping*. *Texture mapping* is a crucial and widely popular technique that uses 2D images to emulate the visual effects of complex 3D geometric details. Each vertex of texture-mapped triangles comes with a 2D texture coordinate, in addition to its 3D coordinate. The most popular texture mapping hardware implementation is based on *mip-mapping*, which pre-calculates reduced-resolution versions of a texture image, and performs weighted filtering between two adjacent resolution levels, which are chosen according to texture coordinates, to arrive at final pixel values.

Finally, the majority of graphics accelerators support depth-buffer based hidden surface removal method for opaque objects. After scan-conversion of a triangle into a set of pixels, each pixel goes through the *depth test* in *pixel processing* stage. This test compares the depth value of the current pixel against the depth value of the pixel at the corresponding X-Y coordinate of the frame buffer. If the new value is smaller, the current pixel is closer to the eye than the old pixel, and therefore, the depth and color value of the current pixel replace the old values. Otherwise, the new pixel values are discarded. For transparent objects, the colors of old and new pixels are composited according to their transparency, or alpha value. This composition is known as *alpha blending*.

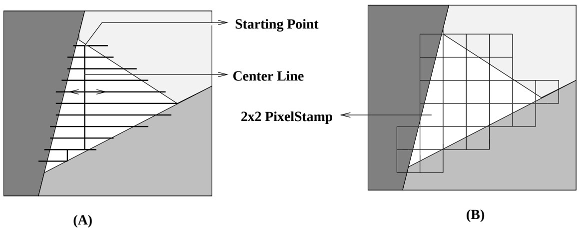


Figure 2: *Scan Conversion of a Triangle using Linear Edge Function.*

3 Methodology

In this section, we describe the benchmarks used for this characterization study, and the tracing and software simulation environment.

3.1 Workload Characteristics

To characterize the architectural aspects of 3D graphics applications, three sets of benchmarks are used: Viewperf OpenGL performance evaluation benchmarks [Gro], a complete time-demo of QuakeII, a popular interactive 3D game from ID software [ID], and three VRML 1.0 [Pes95] models. Viewperf suite consists of five different models: 1) human body from Advanced Visualizer (AWadvs), 2) plant model from Design Review (DRV), 3) scientific visualization data from Visualization Data Explorer (DX), 4) parliament building model from Lightscape (Light), and 5) car model from ProCDRS (CDRS). The benchmark provides a set of tests for each model. The test, that would stress the rendering engine more than others is chosen for each model (Test 1 for AWadvs, DRV, and DX, Test 4 for Light, and Test 6 for CDRS). Each test is run for 100 frames. The VRML models chosen are model of Graz University of Technology, Austria (Graz) [Ins], consisting of sparsely located buildings and huge terrains, an indoor scene of Austrian National Library (Library) [Ins], and model of Aztec city (Aztec) [Sil]. Table 1 shows the general characteristics of the benchmarks. [1] The Viewperf characterization results are reported, whenever possible, because Viewperf models are bigger than other benchmarks. However, some characterization studies are either not applicable, or uninteresting for Viewperf benchmarks. Results for QuakeII and VRML models will be reported in those cases.

3.2 Instrumentation Setup

Our tracing environment consists of Pentium II processor running Linux, with PCI based 3Dfx Voodoo2 card as the 3D accelerator. The 3D graphics software system consists of Mesa-3.0 [Pau], which is a public-domain 3D graphics rendering library implementing OpenGL API [NDW93]. Mesa communicates with Voodoo using Glide rasterization library [3df], that serves as a software micro-layer to the 3Dfx family of cards. We instrumented Mesa to trace all OpenGL calls made

[1] OpenGL primitive GL_TRIANGLES, consisting of multiple triangles is enumerated as multiple primitives. For QuakeII total primitives represent the average over all frames.

by the benchmarks. The interaction between the geometry transformation stage (Mesa) and the rasterization stage (Voodoo2) is captured by tracing the calls made to the Glide library from Mesa. The *linear interpolation* based scan conversion processor is simulated using the software rasterization code that comes with Mesa. We implemented our own software simulator for *linear edge function* based processors.

DataSet	Frames	Total Primitives	Vertex/Primitive	Resolution	Texture
AWadvs	100	18,437	3.39	720x720	YES
DRV	100	21,526	7.03	700x700	NO
DX	100	976	95.84	700x700	NO
Light	100	450,029	4.00	1248x960	NO
CDRS	100	75,136	3.74	1260x980	YES
QuakeII	352	4,675	5.22	640x480	YES
Library	128	10,292	3.38	667x500	YES
Graz	143	8,859	3.18	667x500	YES
Aztec	58	63,933	3.24	667x500	NO

Table 1: *General characteristics of the workloads.*

4 Results and Analysis

4.1 Bandwidth Requirements for Geometry Information

The majority of PC-class 3D graphics boards implements only the rasterization function, leaving the geometric transformation to the host CPU. Therefore, transformed triangles need to be transferred between the CPU and the graphics board, over a high-speed system bus such as PCI. We measure the average bandwidth requirement to send geometry information for a single frame. The results are shown in Figure 3. There are two measurements for each workload, represented as two bars. The first bar shows the bandwidth requirement when the geometry information is sent over as *triangle meshes*, and the second bar for individual triangles. Although the rasterization unit for most 3D graphics cards is triangles, real-world geometric primitives are mostly represented as *triangle mesh*, which is a sequence of triangles, where each triangle is adjacent to its immediate predecessor. *Triangle meshes* allow shared vertices among adjacent triangles to be represented once, and therefore, can lead to significant reduction in bus bandwidth. For Viewperf benchmarks, the mesh representation does not reduce the bandwidth requirement significantly because the number of vertices per geometric primitive is quite small to have sufficient shared vertices (refer Table 1). To render these applications at 30 frames per second, the total bandwidth required is as high as 250 MBytes/sec, which far exceeds PCI's capability. This measurement demonstrates the need for graphics cards to support other compact forms of geometry compression to further reduce the bus bandwidth requirement.

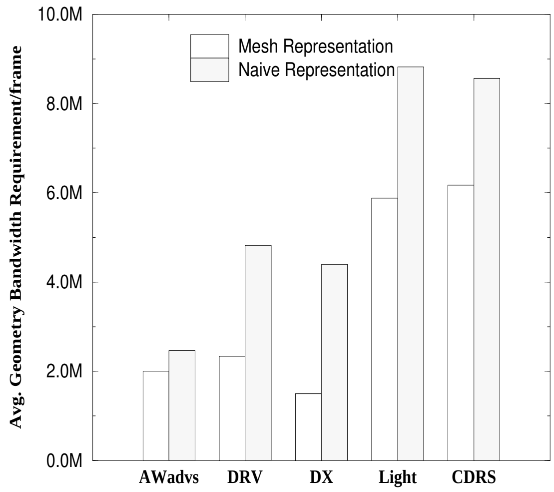


Figure 3: *Average single-frame geometry bandwidth requirement in MBytes for different 3D applications. The first bar shows the bandwidth requirement for triangle mesh representation. The second bar shows the bandwidth requirement for individual triangle representation. Each vertex can have x,y,z coordinates, each in 4 byte floating point, r,g,b for color and alpha for transparency, each 1 byte, and texture coordinates u,v,w, each in 4 byte floating point. Note that a vertex can have a subset of all these attributes.*

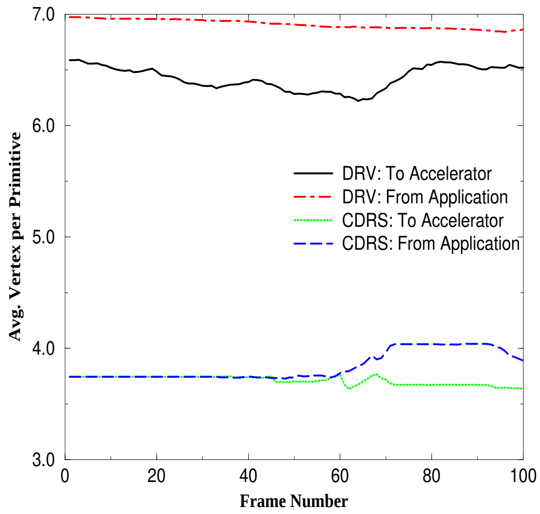


Figure 4: *Average number of vertices per primitive over different frames, which correspond to different viewangles. "From Application" shows the vertex per primitive ratio as defined by the application. "To Accelerator" shows the same ratio after clipping, i.e. the ratio observed by the graphics accelerator.*

As the viewpoint changes for an application, the transformed triangles of the original 3D model goes through a clipping stage, that eliminates the triangles outside the viewing volume. When a constituent triangle of a triangle mesh is clipped, the geometry transformation software (Mesa) traditionally converts the triangle mesh to individual triangles. We measure average vertex per primitive ratio across viewpoints due to this clipping process. Only *Light*, *DRV*, and *CDRS* need clipping, since other two models remain entirely inside the viewing volume. Clipping has negligible effect on *Light* because it has only four-vertex primitives. Figure 4 shows that the vertex per primitive ratio after clipping, is consistently lower than the ratio before clipping for *DRV* and *CDRS*. Because this ratio is low in the first place in the original model for these applications, clipping can not reduce it significantly. However, we think that for 3D models with a large average number of vertices per primitive, the degeneration of triangle meshes into individual triangles due to clipping might cause serious performance problems.

4.2 Rasterization Processing Requirements

Figures 5 and 6 show the triangle and pixel processing requirements per frame over time, respectively. These processing requirements change substantially over time for *Light* and *CDRS*, the walk-through applications. Notice that there is no obvious correlation between the triangle processing and pixel processing demands for the same viewangle. This indicates that average triangle size changes significantly across frames. Figure 7 shows the computation requirements on the *edge processing* stage for different viewpoints. Notice that for *Light*, the number of spans vary considerably across frames; while pixel processing requirement remains comparatively unchanged. This suggests that the change in average span length for *Light* is inversely proportional to the change in the number of spans, resulting in a wide variance of triangle aspect ratio (width/height), for different viewpoints. Finally, Figure 8 shows the total number of *pixelstamps* processed per frame. This curve closely follows the pixel processing requirement, as expected. These figures imply that the relative processing demand placed on different pipeline stages of the rasterization engine changes significantly across viewpoints. The graphics processor needs to have sufficient FIFO buffers between different stages of the pipeline to absorb this variation without introducing stalls.

4.3 Choice of Pixelstamp Granularity

Statistics that affect the design of the scan conversion stage for *linear interpolation* based processors are the distributions of triangle size, span length, and number of spans, which were reported in Chiueh et. al. [CL97]. In this work, we address the design issues for *linear edge function* based scan conversion hardware. The most important design parameter is the size of the pixelstamp. The tradeoff is more parallelism with bigger size versus the wasted processing power due to out-of-boundary pixels. When the triangle is small, a big pixelstamp often overlaps with triangle edges resulting in a reduced pixel throughput [ABM88]. We define *pixelstamp efficiency* as the ratio of the size of the triangle, and the number of pixels at which the edge functions are evaluated. Table 2 shows the average pixelstamp efficiency and average number of pixelstamps required per frame if 2×2 pixelstamps are used. Figures 9 and 10 show the pixelstamp efficiency and the average number of pixelstamps generated for different pixelstamp granularities. We did not consider pixelstamp size beyond 4×4 since most triangles are smaller than 16 pixels. All the numbers are normalized with respect to the case using the 2×2 pixelstamp. In general, pixelstamp efficiency does not depend on its aspect ratio. However, even with 4×4 pixelstamp, the efficiency decreases significantly without any corresponding considerable reduction in number of pixelstamps.

4.4 Depth Test

An important statistics for the pixel processing stage is the percentage of pixels that succeed the *depth test*, and hence participates in costly Read-Modify-Write operation of the frame buffer as opposed to the simpler read operation. The depth test success rate varies from as low as 46% to as high as 85% for the applications we studied, as shown in Table 3. A related statistic is the average

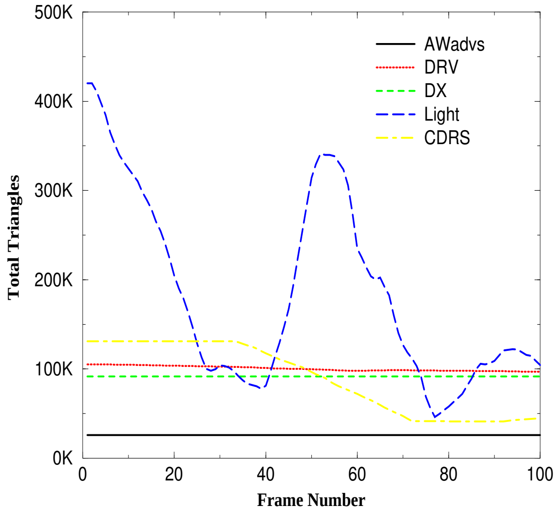


Figure 5: *Total number of triangles processed by the rasterization engine at different frames or viewangles.*

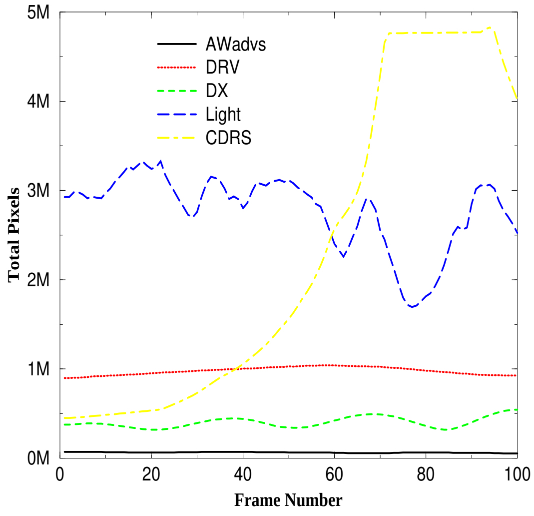


Figure 6: *Total number of pixels processed by the rasterization engine at different frames or viewangles.*

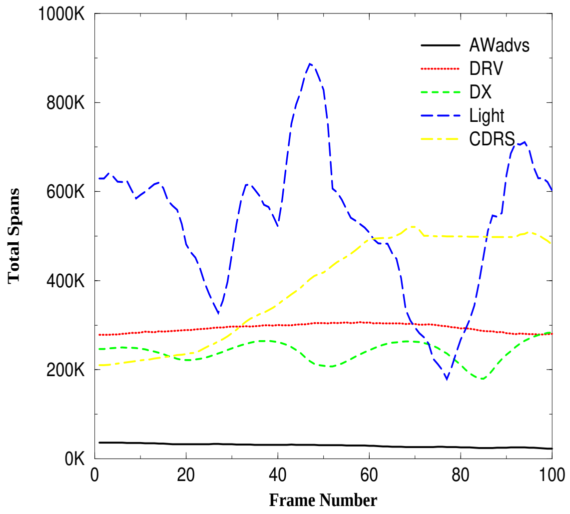


Figure 7: *Total number of spans processed by the rasterization engine at different frames or viewangles.*

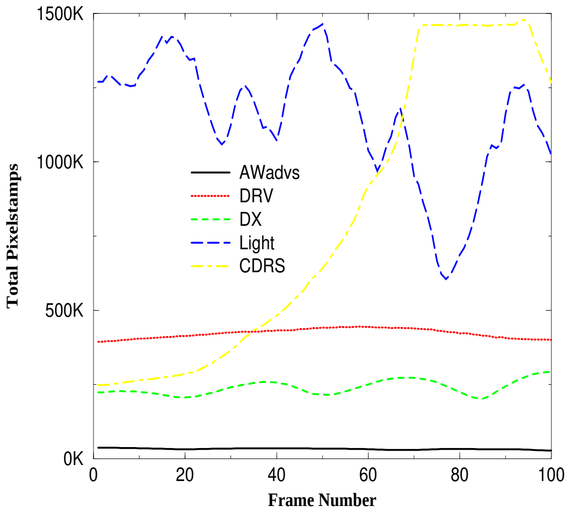


Figure 8: *Total number of pixelstamps processed by the rasterization engine at different frames or viewangles.*

Metric	DataSet				
	AWadvs	DRV	DX	Light	CDRS
Pixelstamp Efficiency	48.23%	57.85%	41.72%	60.90%	63.39%
No. of Pixelstamps	33.00K	422.61K	239.35K	1149.02K	804.33K

Table 2: *Total number of pixelstamps and pixelstamp efficiency for* 2×2 *pixelstamps.*

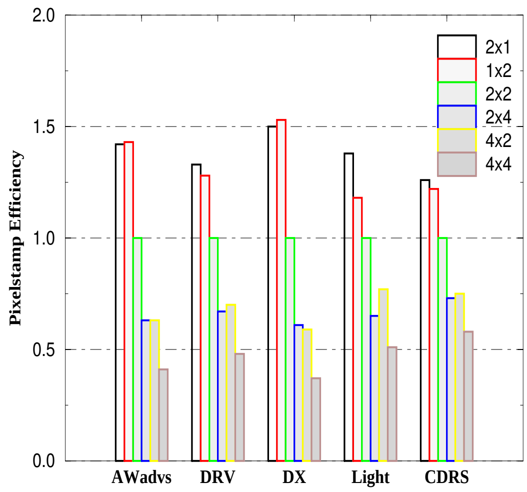


Figure 9: *Pixelstamp efficiency for different pixelstamp granularities. Pixelstamp efficiency is the average ratio of triangle size and total pixels at which edge functions are evaluated. All numbers are normalized with respect to the case using* 2×2 *pixelstamps.*

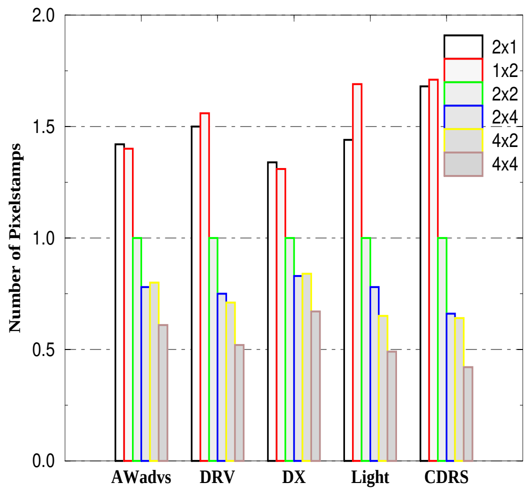


Figure 10: *Total numbers of pixelstamps processed for different pixelstamp granularities. All numbers are normalized with respect to the case using* 2×2 *pixelstamps.*

depth complexity, which measures the average number of triangle pixels that map to a frame buffer pixel. This statistic is particularly useful in load balancing parallel 3D graphics renderers. It is interesting to note that there is no correlation between depth complexity and depth test success rate. This is probably because the triangles of the input model are sorted according to depth, resulting in high depth test success rate irrespective of depth complexity.

Metric	DataSet				
	AWadvs	**DRV**	**DX**	**Light**	**CDRS**
Depth Complexity	1.42	3.35	3.43	2.45	3.30
Depth Test Success %	84.55%	51.10%	56.88%	70.84%	45.89%

Table 3: *Depth test success rate and average depth complexity*

4.5 Memory Access Characteristics During Rasterization

The frame buffer memory bandwidth is a crucial design consideration for the pixel processing stage of the graphics processor. To achieve a high pixel processing throughput, the graphics processor should perform fast access to the frame buffer memory. Current generation of dynamic random access memories (DRAM) can support high throughput so long as the access stream has significant spatial locality. A DRAM array is divided into number of page. A page has to be first activated, or loaded into sense amplifiers, which serve as a cache for DRAM array, before it can be accessed [IBM]. After data has been accessed it has to be written back to the DRAM array, and the array has to be precharged before the next operation. Both the activate and precharge commands are expensive. Hence, the graphics processor should read as much data as possible from a page before moving on to another page [$\mathrm{M}^{+}98$]. We measure the average number of consecutive pixel access per memory page for different mappings of the frame buffer to DRAM pages. The results are shown in Figure 11. Because there is no reason to expect that the triangles would extend into the X or Y axis, it is advantageous to map a rectangular portion of the frame buffer to each page. We considered page size of 256 pixels and 512 pixels with different aspect ratios. The spatial locality is quite small for all the workloads. We used *linear interpolation* algorithm for scan conversion, which favors wide pages, and therefore the page with aspect ratio 1×1 does not always perform better than pages with an asymmetric aspect ratio.

Most DRAMs used in modern graphics cards have more than one internal memory banks [IBM]. If memory accesses alternate between the two banks, the activate and precharge overhead can be hidden. Specifically, once a page of a bank is closed, the pixel access should not return to that bank before it is precharged and ready for the next access. Memory pages of the frame buffer should be mapped to the memory banks in a checker-boarded fashion as shown in Figure 15, to mask the bank switching delay [$\mathrm{M}^{+}98$]. Figure 13 shows the average number of pixels processed between two adjacent but non-consecutive accesses to the same bank. The Synchronous DRAM (SDRAM) or Synchronous Graphics DRAM (SGRAM) used in graphics accelerators have 2 or 4 memory banks [IBM]. The next generation of DRAM architectures, like SLDRAM [GV97] and Direct

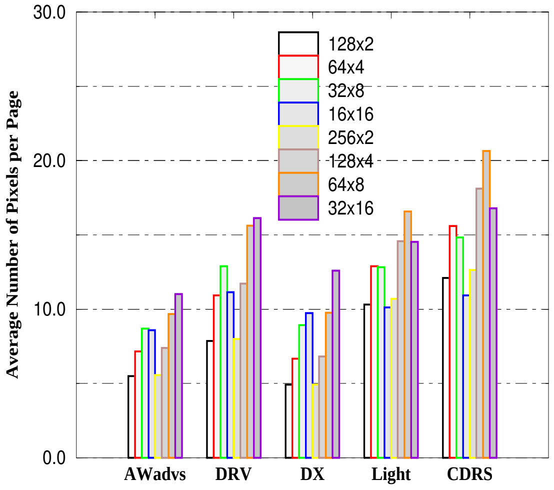


Figure 11: *Average number of consecutive pixel accesses that go to a frame buffer page before switching to another, using* **scanline ordering**. *The numbers are for 256-pixel and 128-pixel page sizes with different aspect ratios.*

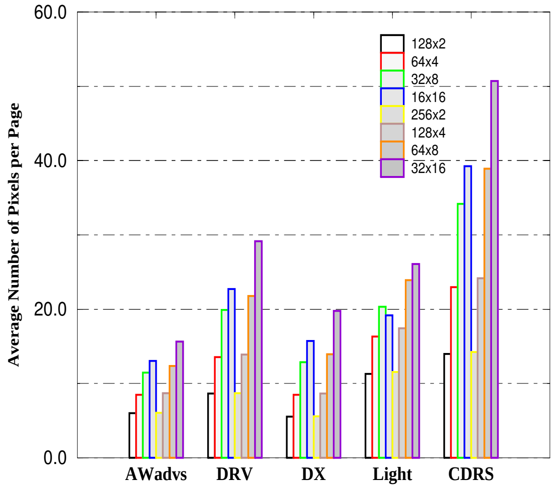


Figure 12: *Average number of consecutive pixel accesses that go to a frame buffer page before switching to another, using* **chunking**. *The numbers are for 256-pixel and 128-pixel page sizes with different aspect ratios.*

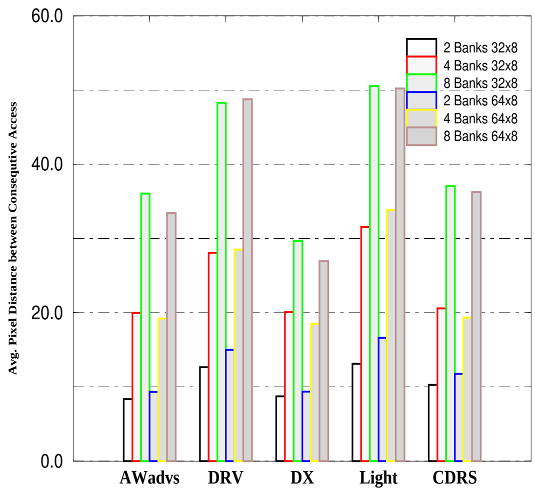


Figure 13: *Average pixel distance between two adjacent but non-consecutive accesses to the same memory bank. This number represents the time available to precharge and activate a memory bank.*

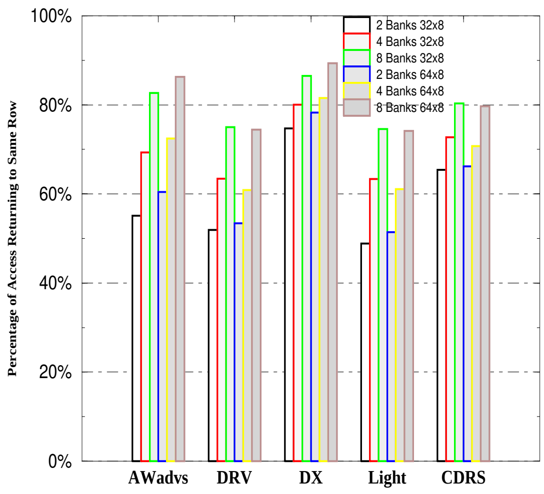


Figure 14: *Percentage of time pixel access returns to the same page of a memory bank after accessing other banks in between.*

2 Banks

0	1	0	1
1	0	1	0
0	1	0	1
1	0	1	0

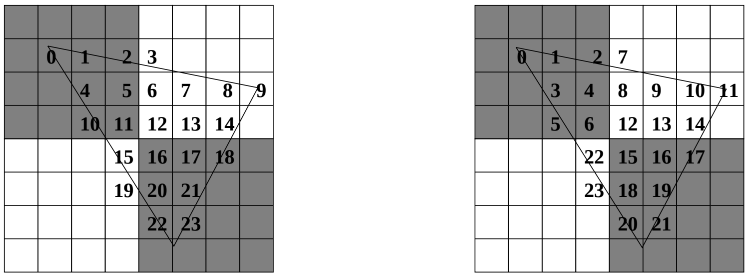
4 Banks

0	1	2	3
2	3	0	1
0	1	2	3
2	3	0	1

8 Banks

0	1	2	3	4	5	6	7
2	3	4	5	6	7	0	1
4	5	6	7	0	1	2	3
6	7	0	1	2	3	4	5

Figure 15: *Checkerboard Page Interleaving that maps pages of the frame buffer to memory banks. The squares represent pages, and the numbers represent the internal memory banks to which the pages are mapped*



Scanline Pixel Generation Order **Chunked Pixel Generation Order**

Figure 16: *Scanline-based versus Chunked pixel generation used during scan conversion. The squares represent pixels and* 4×4 *square is a memory page. The shaded pages belong to internal bank 0, and white pages to bank 1.*

RDRAM [Cri97] have 8 or 16 internal memory banks. The pixel distance increases significantly as the number of banks increases. However, increasing the page size does not seem to affect the pixel distance.

Another important issue for the memory controller is to decide whether to keep a page open on a particular bank when the access stream moves to some other bank [Sch97]. If the access stream is going to return to the page that it just left, keeping the page open can avoid the activate latency. Figure 14 shows the average percentage of references for which this *lazy page close* scheme can eliminate unnecessary activate delays. For the 2-banks case, on an average this percentage is around 50%. However, for 4 or more banks, access returns to the same page most of the time. This is advantageous for the memory controller. When the number of banks is smaller, the number of pixels processed between adjacent but non-consecutive access to the same bank is quite small. If the graphics processor maintains a small FIFO buffer for the pixel addresses, the memory controller can find out the page address of the future access to a bank by looking ahead in the FIFO buffer, and can take the decision of keeping a page open or closed. For a large number of banks, this will require a very large FIFO, and is thus not feasible. However, in that case, since most of the time access returns to the same page, the memory controller can simply choose to keep previous active pages open without incurring much penalty.

One way to improve the locality of frame buffer access is to change the pixel generation order during scan conversion to reduce the number of page crossings. This idea has been implemented in Neon [M$^+$98] and is known as *chunking* (Figure 16). During scan conversion of a triangle, all the pixels belonging to a particular memory page are processed first before moving on to the next page. Also the pages are traversed in checkerboard fashion so as to interleave the access to different internal banks. Figure 12 shows the average number of consecutive pixel accesses that go to the same page with chunking. Compared to Figure 11, Figure 12 demonstrates that chunking is an useful optimization.

4.6 Texture Traffic

As texture mapping becomes pervasive in 3D applications, particularly PC games, efficiently moving bandwidth-intensive texture images between host memory and graphics card increasingly becomes a critical design issue. There are two models used in existing graphics architectures: CPU push vs. card pull [CBS98]. Traditionally, the CPU pushes the texture images that are going to be needed by the rasterizer to the graphics card. Since the CPU has no knowledge of the rasterization process, the unit of transfer in the CPU push model is the entire texture image, even when only a small portion of it is actually needed. The problem becomes more complicated if the card does not have enough memory on board to store the entire texture image. With the advent of high performance bus standard from Intel, called AGP, (Advanced Graphics Port) [Int96], which promises very fast host memory access, graphics cards can directly pull the parts of the texture images that are needed without CPU intervention. This model can also waste host memory bandwidth by unnecessarily fetching the same texture data multiple times. Moreover, the card pull model may suffer longer latency because it fetches texture images on a demand-driven basis.

Figure 17 shows the texture memory bandwidth requirement per frame using the card pull model, assuming the graphics card does not have texture cache. (Viewperf benchmarks are not used for this study because the texture data associated with only two models of that suite is quite small) Without texture caching the bandwidth requirement is quite high. Figure 18 shows the texture working set, which is the number of unique texels needed per frame, for the same workloads. This demonstrates the high reuse of texture data, 50 times on an average per texel per frame for *Quake II* whereas texture cache can create only upto $4\times$ reuse efficiency [Kir98]. If the graphics card stores the *entire* texture working set on board, the host memory bandwidth requirement can be reduced significantly.

Table 4 shows the required size of texture memory per frame for the CPU push method. We use three granularities of texture transfer between host memory and the graphics card. First, whenever a texture image is needed, the entire mip-map, including all resolution levels, is transferred to the graphics card. This is the model used in graphics cards using CPU push method. Second, only images at the pair of resolution levels used in interpolation are transferred. Third, only the parts of the images at the two chosen resolution levels that are actually involved in interpolation are transferred. The minimum transfer unit in this model could be a texel block (4 Texel Window) or a single texel. The last two models would require the CPU to dynamically analyze the portions of texture images that are actually needed, and send only those parts to the graphics card before they are needed. This would permit finer-granularity texture data transfer while reducing the texture data access latency by sending texture image data anticipatively. Hence, these models combine the best of both CPU push and card pull methods. The tradeoff among these three models is software analysis complexity and thus additional overhead on the host CPU, and reduction in texture data transfer traffic. Table 4 shows the average number of bytes of texture image data required to render each frame under each of these three access models. Using texel blocks as the minimum texture transfer unit significantly cuts down the texture traffic bandwidth, as well as texture memory size requirement, and yet is amenable to direct hardware implementations [CBS98].

The other way to improve the texture memory access efficiency is to cache texture images for subsequent reuse. Existing 3D applications, particularly PC games, have paid close attention to intra-frame texture locality [HG97]. In contrast, inter-frame texture locality is less explored in the literature, mainly because the amount of graphics card memory is limited. It is interesting to characterize the evolution of the set of texture images and its total size from frame to frame. Such a characterization helps determine the total graphics card memory requirement if inter-frame texture locality is to be exploited to reduce the texture traffic. Figure 19 shows the correlation of texture data required by frames within a window. Here we assume a "texel block" texture access model. Each point on the upper curve represents the maximum texture memory requirement, or the texture working set, for a window with a given number of frames. This curve establishes the minimum memory requirement on the graphics card in order to exploit inter-frame texture locality at the scale of a given window size. The other two curves represent, when the graphics card's memory can indeed hold the texture working set for a given window size, the average amount of new texture data required in each frame, and the amount of texture traffic that could be saved per frame. Compared to the "Texel Block" column in Table 4, which shows the average amount of texture data that needs to be fetched per frame when inter-frame texture locality is ignored, the "Saved Texture Traffic" curve shows that between 87% to 91% of texture traffic could be eliminated if inter-frame texture locality is exploited. Moreover, the amount of memory required to effectively take advantage of inter-frame texture locality is the texture working set for a window of only 2 to 3 frames! Larger windows do not significantly reduce the texture traffic further.

DataSet	**Average Texture Memory/Frame**			
	Hierarchy	**Resolution Level**	**Texel Block**	**Texel**
QuakeII	1,829,060 (100%)	1,429,278 (78.14%)	328,168 (17.94%)	256,748 (14.03%)
Library	2,293,719 (100%)	144,254 (6.29%)	59,680 (2.60%)	53,990 (2.35%)
Graz	3,155,037 (100%)	164,718 (5.22%)	10,253 (0.32%)	7,945 (0.24%)

Table 4: *The average number of bytes of texture image required to render each frame. Each texel takes four bytes in QuakeII and three bytes in Graz and Library.*

4.7 Predictability of Viewpoint Changes

Some recent graphics cards [Tar98] are designed to work in a multiprocessor configuration. However, multiprocessing alone does not necessarily improve the frame rate because the key input

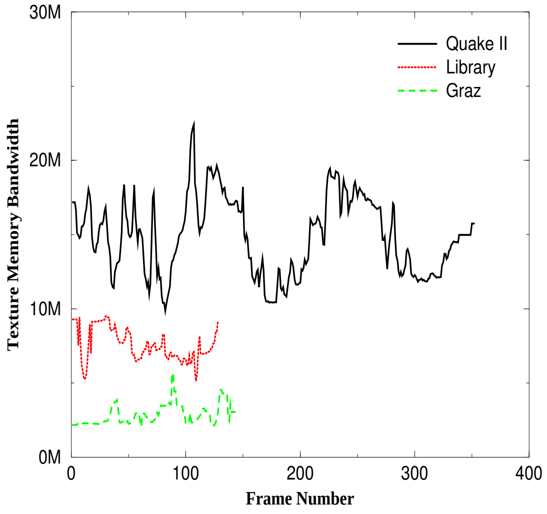


Figure 17: *Total texture memory bandwidth in M-Bytes for different frames.*

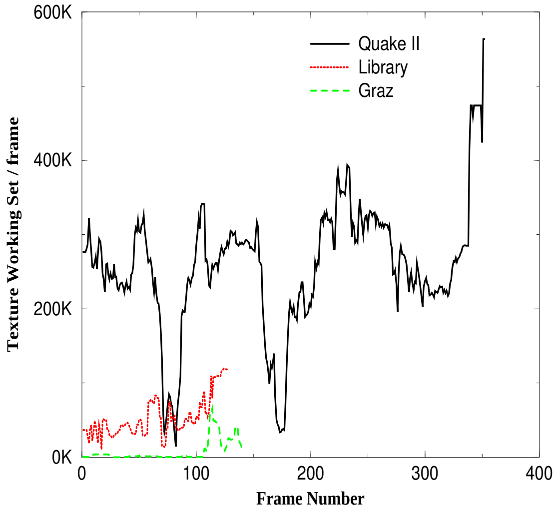


Figure 18: *Texture working set size in KBytes for different frames.*

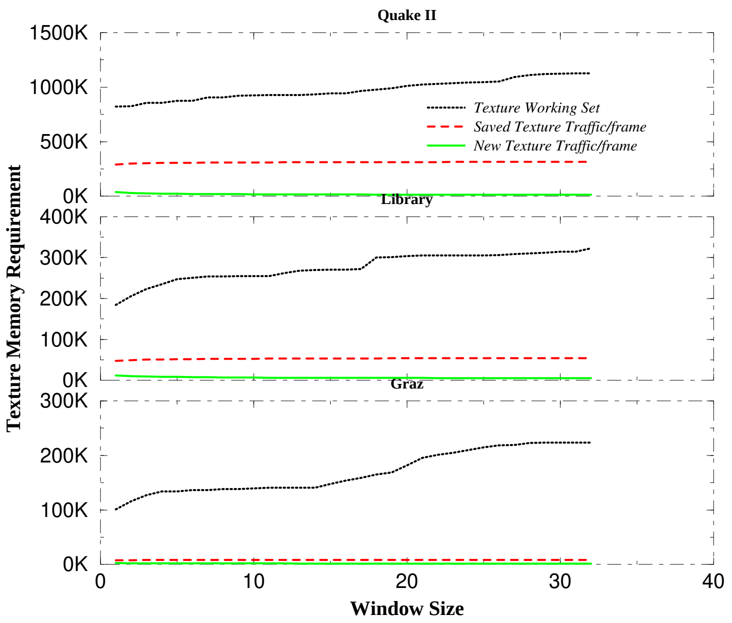


Figure 19: *The inter-frame texture locality measurements for windows of different sizes. The texture working set represents the maximum texture memory requirement for a window with a given number of frames. Saved texture traffic/frame and New texture traffic/frame represent the average texture bandwidth saved, and new texture data required per frame if the texture working set can be stored on board for different window sizes.*

parameter to each frame, the viewpoint, is generally not known in advance. As a result, the frame rate is strictly determined by the rendering latency of individual frames. One way to break this dependency is *speculative rendering* [Ban96], which applies multiprocessing to render subsequent frames based on predicted viewpoints, and uses the speculatively rendered images if the predicted and actual viewpoints fall within a certain proximity. Predicted viewpoints are typically derived from the past trajectory of viewpoints, through first-order or second-order interpolations. When proven successful, speculative rendering fundamentally changes a latency problem into a throughput problem.

The success of speculative rendering depends on how accurately the graphics subsystem can predict the trajectory of the viewpoints and the evolution of viewangles. In this case, we use a simple linear extrapolation based on the viewangles/viewpoints from the last 3 frames to predict the next viewangle/viewpoint. We obtained these statistics by tracing the viewpoints for 2000 frames for both *Graz* and *Library*. We could not collect the statistics for *Aztec* since it does not allow smooth walk-through in our PC based hardware, and for Quake II since the source code is not available. We do not use Viewperf as it does not allow interactive walk-through. Table 5 shows the percentage of frames in which the difference between the predicted and actual viewangles/viewpoints fall within a certain range, which is expressed as a percentage of the viewangle/viewpoint changes from one frame to the next. This simple form of extrapolation achieves a modest prediction accuracy, and demonstrates the predictability of viewpoint/viewangles. However, more sophisticated prediction schemes need to be developed before it is feasible to apply multiple independent hardware engines to improve the interactivity in highly dynamic 3D applications.

4.8 Rendering Pipeline State Changes

Most existing PC-class 3D graphics cards implement a state machine that maintain the rendering parameters to streamline the processing. However, whenever there is a change in texture, shading, or lighting modes, the pipeline state needs to be flushed. The performance overhead of a rasterization pipeline flush could be as small as the cost of rendering a small triangle as in 3Dfx's Voodoo, or up to hundreds of clock cycles. Table 6 shows the average number of triangles in the rasterization stage between consecutive pipeline flushes. Again the statistics are not collected for Viewperf because the benchmarks do not change any rendering parameter from its initialization value.

DataSet	**Prediction Error Percentage**					
	0.0%	**0.1%**	**0.5%**	**1%**	**5%**	**10%**
Graz	0.15%	2.01%	8.13%	11.93%	34.60%	46.68%
Library	0.09%	13.68%	22.59%	24.75%	40.59%	48.42%

Table 5: *Percentages of frames in which the difference between the predicted and actual viewangles fall within a certain range, which is represented as a percentage of the viewangle change from one frame to the next.*

Metric	DataSet			
	Quake II	**Graz**	**Library**	**Aztec**
Flush Distance	72.62	21.21	50.11	578.99

Table 6: *The average number of triangles between consecutive rasterization pipeline flushes because of rendering parameter changes.*

4.9 Dynamic Tiling

Dynamic tiling, as advocated in Microsoft's Talisman Initiative [TK96], attempts to exploit inter-frame coherence to reduce the rendering computation load and thus to increase the frame rate. The most general form of dynamic tiling is to apply a transformation to a portion of the i-th frame to arrive at a certain part of the $i+1$-th frame. In other words, certain parts of the $i+1$-th frame are not rendered from the polygonal 3D model, but from the i-th frame directly. The assumption here is that 2D transformation is in general more efficient than 3D rendering. This technique is related to texture mapping in that the texture image is generated dynamically in this case. Two key design issues in dynamic tiling are *when* and *how* to apply it. In its simplest form, dynamic tiling reuses a portion of the i-th frame directly in the $i+1$-th frame without any transformation, when the distance between the viewpoint and the graphics primitives contributing to that image portion exceeds a certain threshold.

Although dynamic tiling [TK96] is a promising technique for increasing the rendering frame rate, its effectiveness has never been evaluated quantitatively. In view of the significant hardware changes required to support dynamic tiling, it is important to establish an empirical bound on the potential performance improvements from dynamic tiling on real 3D graphics workloads. Graphics architects can then use these bounds to make performance/complexity tradeoffs. The most crucial statistic of interest is the percentage of triangles required to render a frame to which dynamic tiling is applicable. Naturally this percentage statistic depends on the criterion used to qualify "applicable" triangles. Considering that hardware has to make this qualification decision quickly at run time, we use the following two criteria for this characterization study. One can use dynamic titling to render a graphics object if it already appeared in the previous frame and (1) if the normalized distance between the graphics object and the viewpoint exceeds a threshold or (2) if the normalized distance between the graphics object and the viewpoint exceeds a threshold and the area contributed by the graphics object to the final image is smaller than a second threshold. Although graphics objects could be as fine-grained as individual triangles, we choose to use high-level OpenGL objects, as represented by commands embedded between a "glBegin" and "glEnd," as basic units when applying dynamic tiling. We could not collect the statistics for Quake II since it was not possible to match objects from one frame to the other with high accuracy without the source code.

Table 7 shows the average percentage of triangles to which dynamic tiling is applicable for the test workloads under different threshold values. We apply the qualification criteria only to objects in the immediately preceding frame because it is unlikely that there will be a great deal of correlation between objects in two non-consecutive frames. Even if there is, it is expensive for the hardware to capture and exploit it. The *Same Image* column in Table 7 represents those OpenGL objects that have the same rendered image in two consecutive frames. For those objects dynamic tiling is definitely applicable regardless of their normalized distance or area. This captures as high as 13.44% of *Graz*'s rasterized triangles, a modest 8.04% of *Library*'s and almost none of *Aztec*'s. Next we consider the objects that are qualified by the normalized distance and area criteria Table 7 shows 1.92% to 44% of rasterized triangles qualify under different normalized distance thresholds, whereas only 0.10% to 9.79% of them qualify if both the distance and area criteria are used.

While the conventional 3D computation required to render each "qualified" triangle is eliminated, the actual performance improvement of dynamic tiling depends on its run-time implementation cost. In this study, we assume this implementation cost is zero, and thus the result reported here represents the best-case scenario. In addition, we focus only on the performance improvement in the rasterization stage, because detecting "qualified" triangles requires geometric transformation. Table 8 shows the amount of pixel processing that will be saved if dynamic tiling was applied under different criteria. In most cases, the pixel-based percentage savings are half of the corresponding triangle-based percentage savings. This implies the qualified triangles are smaller in size after transformation, which is expected as dynamic tiling mainly targets at distant objects.

DataSet	**Percentage of Qualified Triangles in Rasterizer**						
	Same	**D > 0.9**		**D > 0.95**		**D > .99**	
	Image	**Any A**	**A <= 1**	**Any A**	**A <= 1**	**Any A**	**A <= 1**
Graz	13.44%	17.54%	2.59%	11.25%	1.80%	1.92%	0.10%
Library	8.04%	44.17%	2.88%	25.18%	2.07%	5.29%	0.77%
Aztec	0.04%	24.21%	9.79%	16.45%	6.18%	3.33%	1.26%

Table 7: *Effectiveness of dynamic tiling in terms of percentage of triangles to be rasterized that qualify for dynamic tiling using the normalized distance criterion, and the normalized distance and contributed area criterion. D and A represent the normalized distance and number of pixels contributed by the OpenGL primitive under consideration, respectively.*

DataSet	**Percentage of Saving in terms of pixels**						
	Same	**D > 0.9**		**D > 0.95**		**D > .99**	
	Image	**Any A**	**A <= 1**	**Any A**	**A <= 1**	**Any A**	**A <= 1**
Graz	9.69	8.61	0.12	4.41	0.03	0.70	0.01
Library	4.54	17.35	0.06	6.83	0.05	1.37	0.03
Aztec	0.19	5.42	1.28	3.42	0.73	0.36	0.09

Table 8: *Effectiveness of dynamic tiling in terms of percentage of triangle pixels whose rasterization processing is eliminated due to dynamic tiling, using the normalized distance criterion, and the normalized distance. D and A represent the normalized distance and number of pixels contributed by the OpenGL primitive under consideration, respectively.*

5 Conclusion

The major contribution of this research is a detailed workload characterization of representative 3D applications, which relates empirical workload parameters to specific architectural design tradeoffs. This work follows the quantitative tradition of modern computer architecture research by offering measured rendering statistics that could help graphics hardware architects to choose proper design points for various aspects of the rasterization pipeline. More concretely, this characterization study revealed the following:

- The bandwidth requirement for transferring geometry information from host CPU to graphics card requires either higher-performance system interconnects or more efficient geometry compression algorithms for real time rendering (30 frames/sec) of existing 3D applications.
- The majority of triangles sent to the rasterizer are quite small, making it less likely for the pixelstamp approach to be able to achieve higher parallelism using bigger pixelstamps.
- Spatial locality of frame buffer access is low for scanline-based rasterization order. One can hide the performance penalty due to the lack of locality by exploiting internal memory banks of modern DRAMs. However, with increasing pixel processing requirements, block based rasterization order, like *chunking* is a much better alternative as far as improving locality is concerned.
- Neither the CPU push nor card pull method seems to be able to solve the texture memory size, bandwidth and latency problem. Relying on the CPU to actively manage the texture memory required by the rasterizer is a promising hybrid to achieve the best of both techniques.
- The walk-thorough applications seem to have a relatively predictable viewpoint change over time. Speculative rendering could exploit viewpoint predictability to trade throughput for latency.
- Although the performance penalty associated with pipeline flush increases with the pipeline complexity, the performance degradation due to rendering state changes in our tested applications is insignificant.
- Dynamic tiling based on simple distance and triangle size criteria does not appear very effective in reducing the rendering computation requirements.

There are several avenues from this work that we are currently following up on. First, we are extending the scope of this study by incorporating more 3D workloads that could be measured in our instrumentation test-bed, so that we can start building statistical models from the raw statistics. Second, we are investigating specific architectural and implementation mechanisms that could exploit some of the workload statistics described above, so that we can more accurately quantify the performance cost and therefore the net gain of these architectural ideas. Finally, we are implementing some of these architectural mechanisms in the *Sunder* machine, which is an all-software cluster-based parallel 3D graphics engine.

References

[3df] 3dfx Interactive, Inc. *Glide Rasterization Library.* `http://www.3dfx.com`.

[ABM88] B. Apgar, B. Bersack, and A. Mammen. A Display System for the Stellar Graphics Supercomputer Model GS1000. In *Proceedings of ACM Conference on Computer Graphics (SIGGRAPH)*, 1988.

[Ban96] D. C. Banks. The ImageSwitcher: A Proposed System Architecture Designed to Reduce VR Lag. In *Proceedings of Eurographics Workshop on Graphics Hardware*, 1996.

[Bla92] R. Blau. *Performance Evaluation for Computer Image Synthesis Systems.* PhD thesis, Department of Computer Science, University of California, Berkeley, 1992.

[C+98] M. Chen et al. Simple Models of the Impact of Overlap in Bucket Rendering. In *Proceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 1998.

[CB97] M. Cox and N. Bhandari. Architectural Implications of Hardware-Accelerated Bucket Rendering on the PC. In *Proceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 1997.

[CBS98] M. Cox, N. Bhandari, and M. Shantz. Multi-Level Texture Caching for 3D Graphics Hardware. In *Proceedings of ACM/IEEE International Symposium on Computer Architecture(ISCA)*, 1998.

[CH92] M. Cox and P. Hanrahan. Depth Complexity in Object-Parallel Graphic Architectures. Technical Report CS-TR-382-92, Department of Computer Science, Princeton University, 1992.

[CL97] T. Chiueh and W. Lin. Characterization of Static 3D Graphics Workloads. In *Proceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 1997.

[Cri97] R. Crisp. Direct Rambus Technology: The New Main Memory Standard. *IEEE MICRO*, 17(6), 1997.

[DL90] J. C. Dunwoody and M. A. Linton. Tracing Interactive 3D Graphics Programs. In *Proceedings of ACM Symposium on Interactive 3D Graphics*, 1990.

[F+85] H. Fuchs et al. Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes. In *Proceedings of ACM Conference on Computer Graphics (SIGGRAPH)*, 1985.

[FvF+90] J. D. Foley, A. vanDam, S. K. Feiner, J. F. Hughes, and R. L. Phillips. *Computer Graphics: Principles and Practice.* Addison-Wesley Publishing Company, second edition, 1990.

[Gro] OpenGL Performance Charcterization(OPC) Group. *Viewperf: OpenGL Performance Benchmark.* `http://www.spec.org/gpc/opc.static/opcview.html`.

[GV97] P. Gillingham and B. Vogley. SLDRAM: High-Performance, Open-Standard Memory. *IEEE MICRO*, 17(6), 1997.

[HG97] Z. S. Hakura and A. Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In *Proceedings of ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 1997.

[HP96] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc, second edition, 1996.

[IBM] IBM Microelectronics. *Synchronous DRAM: The DRAM of the Future.* `http://www.chips.ibm.com/products/memory/sdramart/sdramart.html`.

[ID] ID Software. *QuakeII*. `http://www.idsoftware.com`.

[Ins] Institute for Information Processing and Computer Supported New Media(IICM), Graz University of Technology, Austria. *Sample VRML models*. `http://www2.iicm.edu/vrml`.

[Int96] Intel Corporation. *Accelerated Graphics Port Interface Specification, Revision 1.0*. `http://www.intel.com/pc-supp/platform/agfxport/agp10.pdf`, 1996.

[Kir98] D. Kirk. Unsolved Problems and Opportunities for High-quality, High-performance 3-D Graphics on a PC Platform. In *ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, `http://www.merl.com/hwws98/presentations/kirk/index.html`, 1998. Keynote Address.

[M+98] J. McCormack et al. Neon: A Single-Chip 3D Workstation Graphics Accelerator. In *Proceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 1998.

[Mac96] B. MacIntyre. *PC 3D Graphics Accelerator FAQ*. `http://www.cs.columbia.edu/~bm/3dcards/3d-cards1.html`, 1996.

[NDW93] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide*. Addison-Wesley Publishing, Inc, 1993.

[Pau] B. Paul. *The Mesa-3D graphics library*. `http://gopher.ssec.wisc.edu/~brianp/Mesa.html`.

[Pes95] M. Pesce. *VRML: Browsing and Building Cyberspace*. New Riders/Macmillan, 1995.

[Pin88] J. Pineda. A Parallel Algorithm for Polygon Rasterization. In *Proceedings of ACM Conference on Computer Graphics (SIGGRAPH)*, 1988.

[Raf77] A. Rafii. *Empirical and analytical studies of program reference behavior*. PhD thesis, Department of Electrical Engineering, Stanford University, 1977.

[Sch97] R. C. Schumann. Design of the 21174 Memory Controller for DIGITAL Personal Workstations. *Digital Technical Journal*, 9(2), 1997.

[Sil] Silicon Graphics, Inc. *Virtual Model of Aztec*. `http://vrml.sgi.com`.

[Tar98] G. Tarolli. Optimizing a Graphics Application for Consumer PC Graphics. In *ACM SIGGRAPH '98 Course No. 29*, 1998.

[TK96] J. Torborg and J. T. Kajiya. Talisman: Commodity Realtime 3D Graphics for the PC. In *Proceedings of ACM Conference on Computer Graphics (SIGGRAPH)*, 1996.

[Whe85] D. S. Whelan. *ANIMAC: A multiprocessor architecture for real-time computer animation*. PhD thesis, Computer Science Department, California Institute of Technology, 1985.