# 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 focuses on interactive 3D applications spanning multiple frames. To the best of our knowledge, this is one of the first such studies that have ever been attempted. 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 the workload impacts graphics architecture features such as rasterization pipeline, frame buffer design, texture memory management and system bus design.*

## 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 generation of applications. An important class of media-rich applications is critically dependent on photo-realistic 3D graphics technology at video-like frame rate. Despite the large amount of industrial efforts spent in recent years on PC-class 3D graphics chips and the impressive strides made by 3D graphics hardware technology [20], 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. Second, most of the earlier research on 3D graphics hardware tends to be less quantitative than qualitative, making it impossible to empirically compare various architectural ideas. 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 [14, 5]. While it is difficult to solve the first problem, we believe that the second problem can be easily addressed 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 [15].

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 graphics processor architecture. Specifically, we focus on *dynamic* (study of rendering dynamics across image frames rather than within individual images), *polygonal* (the basic primitives used are polygons, which is supported by all existing 3D chips) 3D graphics workloads. This study is thus quite different from and much more wide in scope than the study reported in [4]. Most published work related to 3D graphics workload characterization focused either on specific statistics such as object partitioning strategy for multiprocessor rendering [29], depth complexity and its implications on parallel rendering [7], visible surface measurements [3] or on the design and implementation of 3D application tracing environments [9]. 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. Due to space constraints, some results are not included in this paper and can be found in [22], the technical report version of this paper. 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 constitutes 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 [10]: *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/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 [10], 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 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 [11], 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

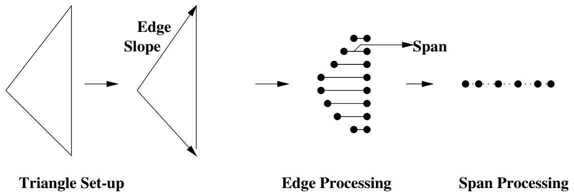


**Figure 1. Scan conversion using linear interpolation.**

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 [26]. 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 tra-

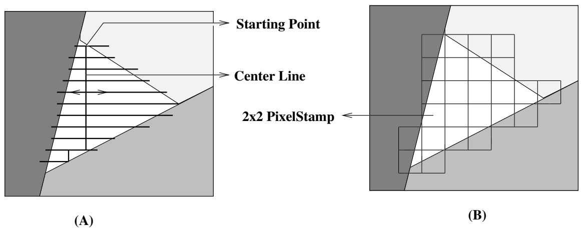


**Figure 2. Scan conversion using linear edge function. (After [26])**

versed one *pixelstamp* at a time, instead of a single pixel [26]. 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 performs *texture mapping*, which is a crucial and widely popular technique that wraps 2D images on 3D objects to emulate the visual effects of complex 3D geometric details. Each vertex of a texture-mapped triangle comes with a texture coordinate, that defines the part of the texture map to be applied to the triangle. The most popular texture mapping hardware implementation is based on *mip-mapping*, which pre-calculates multiple reduced-resolution versions of a texture image. The rasterizer performs weighted filtering between two adjacent resolution levels, which are chosen according to the distance of the object, to arrive at final pixel values.

Finally, the majority of graphics accelerators support depth-buffer based hidden surface removal method for opaque objects. Each pixel generated in the scan-conversion stage goes through the *depth test*, that 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 color of old and new pixel are composited according to their transparency or alpha value. This composition is known as *alpha blending*.

## 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 [13], a complete time-demo of a popular interactive 3D game QuakeII [17], and three VRML 1.0 [25] models. Viewperf suite consists of five scientific or CAD/CAM applications, each with a set of tests: human body (AWadvs), plant (DRV), scientific visualization (DX), parliament building (Light), and car (CDRS). For each application, we chose the test that would stress the rendering engine most ( Test 1 for AWadvs, DRV, and DX, Test 4 for Light, and Test 6 for CDRS). The VRML models chosen are Graz University of Technology (Graz) [18], consisting of sparsely located buildings and huge terrains, an indoor scene of Austrian National Library (Library) [18], and Aztec city (Aztec) [28]. Table 1 shows the general characteristics of the benchmarks. OpenGL primitive *GL_TRIANGLES*, consisting of multiple triangles is enumerated as multiple primitives. For QuakeII, total primitives represent the average over all frames. Antialiasing was turned off for all the applications. The Viewperf characterization results are reported, whenever possible, because Viewperf models are bigger than other benchmarks. However, if some characterization studies are either not applicable, or uninteresting for Viewperf benchmarks, we report the results for QuakeII and VRML models.

### 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 [24], which is a public-domain 3D graphics rendering library implementing OpenGL API [23]. Mesa communicates with Voodoo using Glide rasterization library [1], that serves as a software micro-layer to the 3Dfx family of cards. We instrumented Mesa to trace all OpenGL calls made 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.

## 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

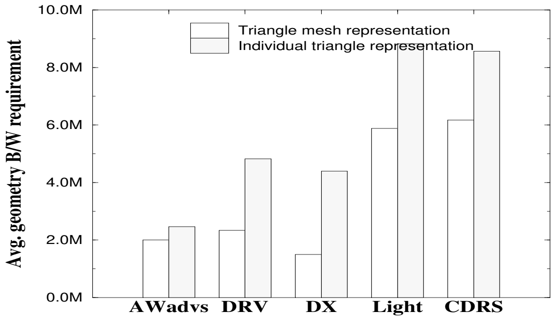


**Figure 3. Average single-frame geometry bandwidth requirement in MBytes.**

Figure 3, where the first bar shows the bandwidth requirement for triangle mesh, and the second bar for individual triangle representation. (Assuming each vertex can potentially have xyz coordinates, each 4 bytes, rgb for color and alpha for transparency, each 1 byte, and texture coordinates uvw, each 4 bytes.) 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

| 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.**

the bandwidth requirement significantly because the number of vertices per geometric primitive is quite small (except for DX) 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.

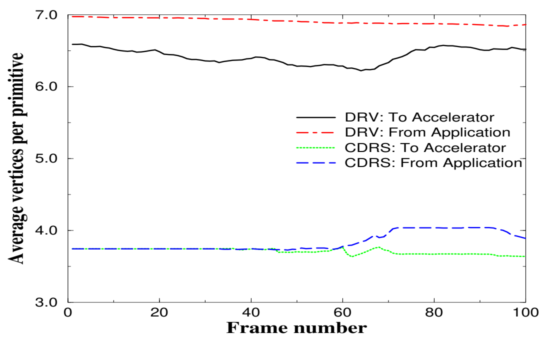


**Figure 4. Average number of vertices per primitive over different frames.**

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. In Figure 4, "From Application" shows the vertex per primitive ratio as defined by the application and "To Accelerator" shows the same ratio after clipping, i.e. the ratio observed by the graphics accelerator. The vertex per primitive ratio after clipping, is consistently lower than the ratio before clipping. 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

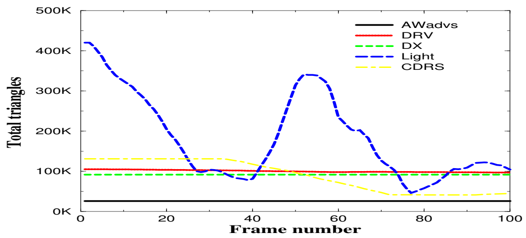


**Figure 5. Total triangles processed per frame.**

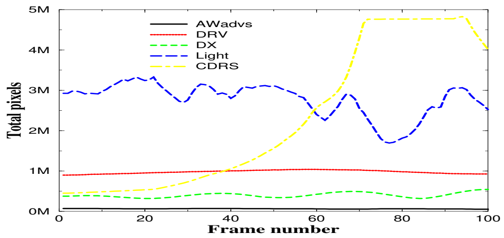


**Figure 6. Total pixels processed per frame.**

Figures 5 and 6 show the triangle and pixel processing requirements per frame, which 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 view-angle. This indicates that average triangle size changes significantly across frames. Figure 7 shows the computation

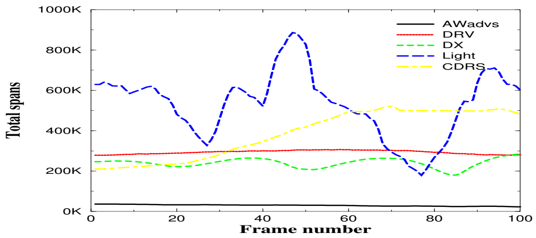


**Figure 7. Total spans processed per frame.**

requirements on the *edge processing* stage for different viewpoints. 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, which 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.

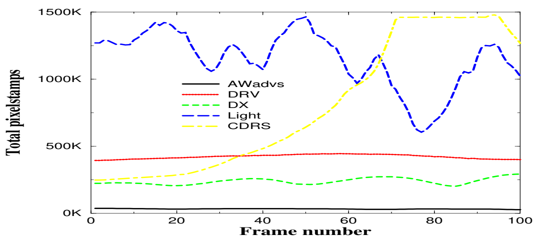


**Figure 8. Total number of $2 \times 2$ pixelstamps processed per frame.**

## 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. [4]. 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 [2]. 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 $2 \times 2$ 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 $4 \times 4$ since most triangles are smaller than 16 pixels. In general, pixelstamp efficiency does not depend on its aspect ratio except for *Light*. This indicates that the datasets do not have many triangles with highly asymmetric aspect ratio. However, even with $4 \times 4$ pixelstamp, the efficiency decreases significantly without any corresponding considerable reduction in number of pixelstamps.

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

**Table 2. Total number of pixelstamps and pixelstamp efficiency for $2 \times 2$ pixelstamps.**

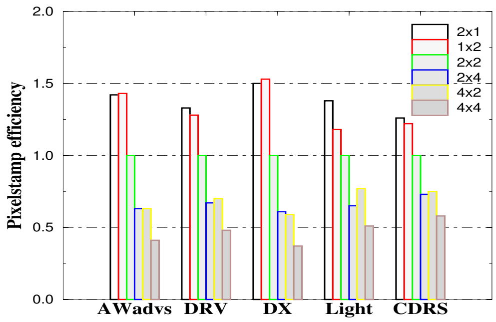


**Figure 9. Pixelstamp efficiency for different granularities normalized w.r.t $2 \times 2$ case.**

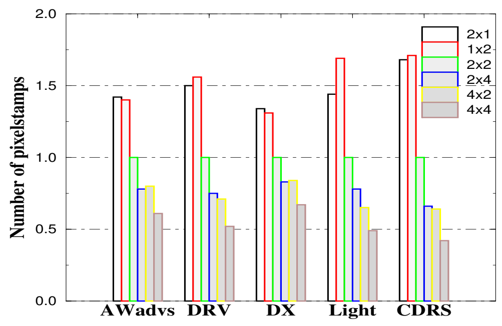


**Figure 10. Pixelstamps processed for different granularities, normalized w.r.t. $2 \times 2$ case.**

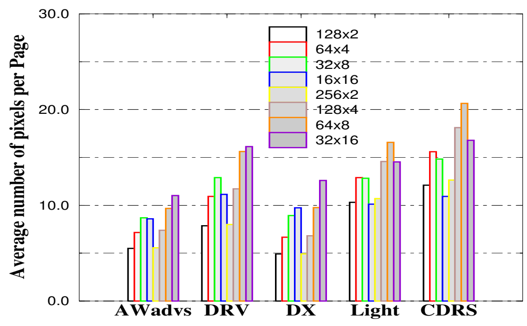


**Figure 11. Average number of consecutive pixel accesses to a frame buffer page using scanline ordering for 256 and 512 pixel page sizes with different aspect ratios.**

## 4.4. Memory access characteristics during rasterization

The frame buffer memory bandwidth and access latency are crucial design considerations for the pixel processing throughput of the graphics processor. 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 pages. 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 [16]. 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 [21]. We measure the average number of consecutive pixel accesses per memory page for different mappings of the frame buffer to DRAM pages (Refer 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. 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 $1 \times 1$ 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 [16]. If memory accesses alternate between the two banks, the 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 12, to mask

**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 12. Checkerboard page interleaving. The squares represent pages, and the numbers represent the internal memory banks to which the pages are mapped (After [21])**

the bank switching delay [21]. Figure 13 shows the average number of pixels processed between two adjacent but non-consecutive accesses to the same bank, i.e., the number of pixels processed by other banks before access returns to the same bank. This indicates the time available to perform precharge operation. The Synchronous DRAM (SDRAM) or Synchronous Graphics DRAM (SGRAM) used in graphics accelerators have 2 or 4 memory banks [16]. The next generation of DRAM architectures, like SLDRAM [12] and Direct RDRAM [8] 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 [27]. If the access stream is going to return to the page that it just left, keeping the page open can avoid the activate latency. Note that, unlike precharge operation, page activation requires the page address, and hence cannot be hidden by simple bank switching. Figure 14 shows that the average percentage of

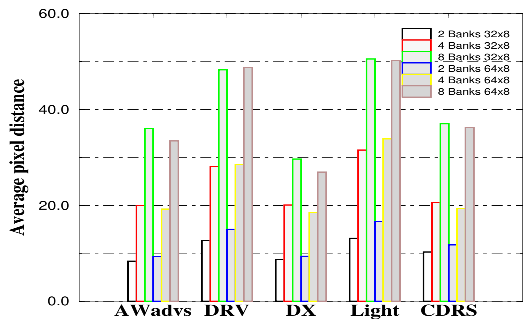


**Figure 13. Average pixel distance between two adjacent but non-consecutive accesses to the same memory bank.**

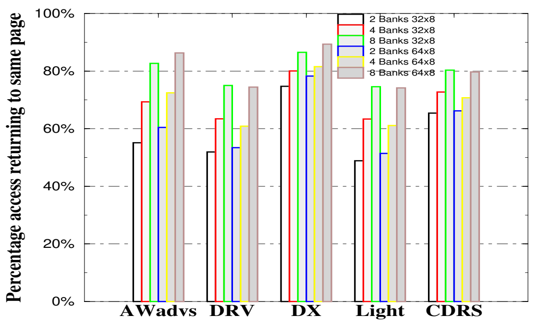


**Figure 14. Percentage of time pixel access returns to the same page of a memory bank after accessing other banks in between.**

references for which this *lazy page close* scheme can eliminate unnecessary activate delays is around 50% for 2 banks and much higher for 4 or more banks. This is advantageous for the memory controller. When the number of banks is smaller, the number of pixels processed for other banks before returning to the same bank is quite small. If the graphics processor maintains a small FIFO buffer for the pixel addresses, the memory controller can take the decision of keeping a page open or closed by looking at the page address of the future access to a bank in the FIFO buffer. 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.

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 [21] and is known as *chunking* (Figure 15). During scan conversion of a triangle, all the pixels

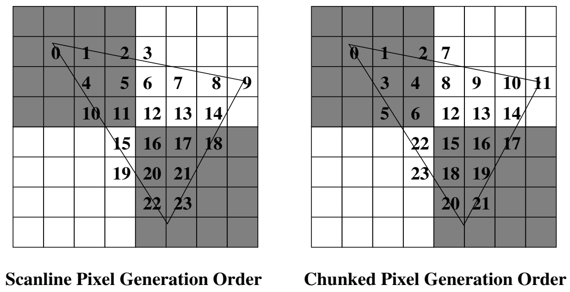


**Figure 15. Scanline versus chunked pixel generation order. The squares represent pixels and $4\times4$ square is a memory page. The shaded pages belong to internal bank 0, and white pages to bank 1. (After [21])**

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. Compared to Figure 11, Figure 16

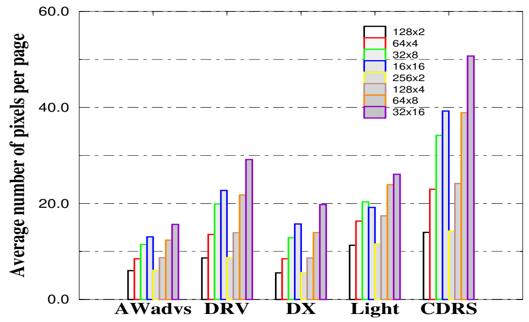


**Figure 16. Average number of consecutive pixel accesses to a frame buffer page using chunking for 256 and 512 pixel page sizes with different aspect ratios.**

demonstrates that chunking is an useful optimization. Note that Figure 11 and 16 have different maximum values for the Y-axis.

### 4.5. Texture traffic

As texture mapping becomes pervasive in 3D applications, particularly games, a critical design issue is efficiently transferring bandwidth-intensive texture images between host memory and graphics card. There are two models used in existing graphics architectures: CPU push vs. card pull [6]. Traditionally, the CPU pushes the texture images required 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) [19], graphics cards can directly pull the required parts of the texture images without CPU intervention. However, this model can also waste host memory bandwidth by unnecessarily fetching the same texture data multiple times. Moreover, the demand-driver texture fetching mechanism of the card pull model may suffer longer latency. Figure 17 shows the texture memory

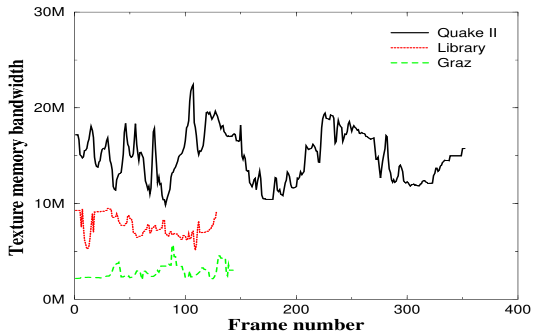


**Figure 17. Total texture memory Bandwidth per frame in MBytes.**

bandwidth requirement per frame is quite high for 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) The texture working set (number of unique texels or texture pixels) in Figure 18, however, is quite small 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 up to $4\times$ reuse efficiency [20]. If the graphics card stores the *entire* texture working set on board, the host memory bandwidth requirement can be reduced significantly.

We use three granularities of texture transfer between host memory and the graphics card for the CPU push model : (1) the entire mip-map, including all resolution levels, (2) the images at the pair of resolution levels used in interpolation, and (3) the parts of the images at the two chosen reso-

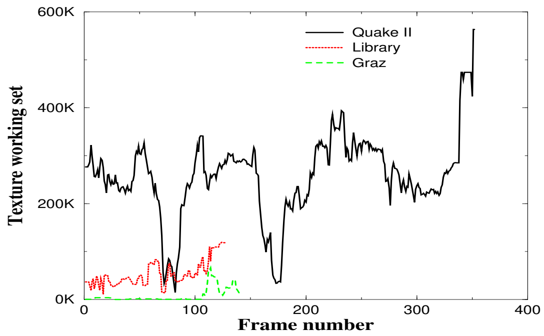


**Figure 18. Texture working set size in KBytes.**

lution levels that are actually involved in interpolation. We choose the minimum transfer unit in the last model as either a $4 \times 4$ texel block or a single texel. The first model is the one used in graphics cards with CPU push method. The last two models would require the CPU to dynamically determine the required portions of texture images. This would permit finer-granularity texture data transfer while reducing texture access latency by sending the images anticipatively. Hence, these models models combine the best of both the CPU push and card pull methods. The tradeoff among these models is the additional overhead on the host CPU, and reduction in texture data traffic. Table 3 shows the average texture memory size required per frame under each of these access models. (Each texel takes four bytes in QuakeII and three bytes in Graz and Library.) 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 [6].

The other way to improve the texture memory access efficiency is to cache texture images for subsequent reuse. Existing 3D applications, and architects have paid close attention to intra-frame texture locality [14]. 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. Figure 19 shows the correlation of texture data required by frames within a window. The upper curve represents the texture working set, i.e. the minimum memory requirement on board in order to exploit inter-frame texture locality, for a window with a given number of frames. 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, and larger windows do not significantly reduce the texture traffic further. The other two curves rep-

| 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 3. The average number of bytes of texture image required to render each frame.**

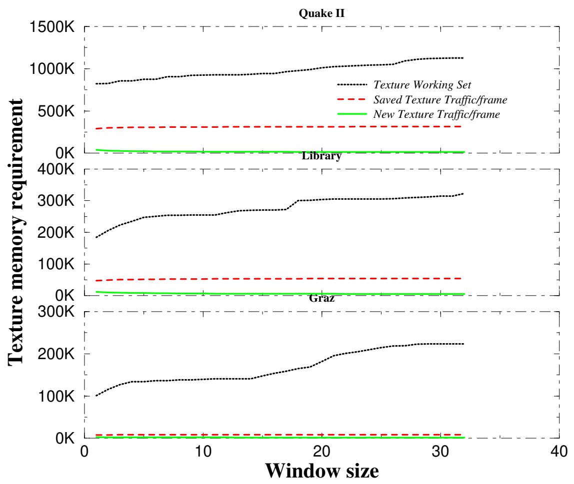


**Figure 19. Inter-frame texture locality for different window sizes with $4 \times 4$ texel block.**

resent, 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 3, 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.

### 4.6. Rendering pipeline state changes

Most existing PC-class 3D graphics cards implement a state machine that maintains 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 4 shows the average number of triangles in the rasterization stage between consecutive pipeline flushes because of rendering parameter changes.

| Dataset | Flush Distance |
|---|---|
| QuakeII | 72.62 |
| Graz | 21.21 |
| Library | 50.11 |
| Aztec | 578.99 |

**Table 4. Average number of triangles between consecutive rasterization pipeline flushes.**

Again the statistics are not collected for Viewperf because the benchmarks do not change any rendering parameter from its initialization value.

## 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 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 of 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. While one can hide the performance penalty due to the lack of locality by exploiting internal memory banks of modern DRAMs, block based rasterization order, like *chunking* is a much better alternative, with increasing pixel processing requirements,

- 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 might be a promising hybrid to achieve the best of both techniques,

- 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.

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. 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 tradeoff between implementation cost and performance gain of these architectural ideas.

## 6. Acknowledgement

This research is supported by an NSF Career Award MIP-9502067, NSF MIP-9710622, NSF IRI-9711635, NSF EIA-9818342, NSF ANI-9814934, a contract 95F138600000 from Community Management Staff's Massive Digital Data System Program, USENIX student research grants, as well as fundings from Sandia National Laboratory, Reuters Information Technology Inc., and Computer Associates/Cheyenne Inc.

## References

[1] 3dfx Interactive, Inc. *Glide Rasterization Library*.

[2] 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.

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

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

[5] 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.

[6] 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.

[7] M. Cox and P. Hanrahan. Depth Complexity in Object-Parallel Graphic Architectures. Technical Report CS-TR-382-92, Dept. of Comp. Sc., Princeton University, 1992.

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

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

[10] 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.

[11] 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.

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

[13] O. P. C. Group. *Viewperf: OpenGL Performance Benchmark*. `http://www.spec.org/gpc/opc.static/opcview.html`.

[14] Z. S. Hakura and A. Gupta. The Design and Analysis of a Cache Architecture for Texture Mapping. In *Proceedings of ISCA*, 1997.

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

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

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

[18] IICM, Graz University of Technology, Austria. *Sample VRML models*. `http://www2.iicm.edu/vrml`.

[19] Intel Corporation. *Accelerated Graphics Port Interface Specification, Revision 1.0*, 1996.

[20] 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*, 1998.

[21] J. McCormack et al. Implementing Neon: A 256-Bit Graphics Accelerator. *IEEE MICRO*, 19(2), 1999.

[22] T. Mitra and T. Chiueh. Dynamic 3D Graphics Workload Characterization and the Architectural Implications. Technical Report ECSL-TR-61, Computer Science Department, SUNY at Stony Brook, `http://www.ecsl.cs.sunysb.edu/tech_reports.html`, 1999.

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

[24] B. Paul. *The Mesa-3D graphics library*. `http://www.mesa3d.org`.

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

[26] J. Pineda. A Parallel Algorithm for Polygon Rasterization. In *Proceedings of SIGGRAPH*, 1988.

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

[28] Silicon Graphics, Inc. *Virtual Model of Aztec*.

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