Characterization of Static 3D Graphics Workloads

Tzi-cker Chiueh Wei-jen Lin

Computer Science Department
State University of New York at Stony Brook
Stony Brook, NY 11794-4400
chiueh@cs.sunysb.edu

Abstract

3D graphics transform 3D models into 2D images by simulating the physics of light propagation from the lighting sources, through the objects, and eventually to the eyes. Although specialized graphics hardware engines have been proposed and implemented in the past, and a heated interest in PC-class 3D graphics cards is currently emerging, detailed descriptions and analysis of 3D graphics workloads which graphics hardware design can be based on are almost non-existent. This work takes the first step towards a comprehensive 3D graphics workload characterization by reporting the results of an empirical study using an instrumented software polygonal renderer tested on a wide variety of static 3D graphics models with sufficiently sophisticated geometric and texture properties.

Keywords: 3D graphics rendering, depth complexity, span size, graphics workload characterization, graphics compression, graphics pipeline

1 Introduction

3D graphics render three-dimensional model databases into two-dimensional images for given viewpoints by simulating the physical interaction between the light sources from specified directions and the objects with complex geometrical and shading properties. The ultimate quality goal of 3D graphics is ***photo-realism***, i.e., when there is no discernible distinction between synthetic images generated by computer graphics systems and photos captured by cameras. Nature creates high-quality images by shedding millions of light rays upon physical objects and waiting for the image to emerge from the underlying physics. 3D computer graphics systems attempt to approximate the same image quality while consuming only a small fraction of the resource that Nature uses. Nevertheless, high-end 3D graphics rendering is still considered highly computationally expensive, even with specialized hardware assists implemented in advanced VLSI technology.

3D graphics rendering methods in general can be classified into two types: volume and surface rendering. When the interior properties of objects are of interest, such as biomedical (Ultrasound, Computer Tomography, Magnetic Resonance) imaging, volume rendering usually is the method of choice because its rendering primitive is a small 3D point called a voxel. Voxels are much smaller than geometric primitives used in surface rendering, which typically only focuses on object surfaces, and therefore allow more comprehensive sampling of the object space. On the other hand, volume rendering is almost always computationally more expensive than surface rendering. Most of the 3D graphics systems used today in applications such as engineering design, entertainment, and advertising are based on surface rendering. Among surface rendering methods, polygon graphics refer to those methods that use planar polygons as the only geometric primitives during rendering. Non-polygonal graphics can use procedural specifications for object geometry such as parametric curves and surfaces, fractals and particle systems. Most 3D graphics hardware engines use polygonal surface rendering, which is also the focus of this work.

Although the development of 3D graphics hardware started almost as early as the 3D graphics itself, most of these engines, notably SGI machines, have until recently remained out of reach for everyday applications because the special software and hardware required are cost prohibitive for PC-class users. Since the middle of 1995, a flurry of PC-based 3D graphics cards [7] started to flood into the marketplace. Most of these cards came from vendors that were previously manufacturing 2D video cards. The first generation of these 3D graphics boards did not perform as well as expected. However, subsequent generations are getting better and better, and start to pose serious threats to high-end graphics machines. As of October 1996, some of these cards already exhibit respectable performance, e.g., four ISP chips from NEC/PowerVR together can perform 1,028K mip-mapped textured, smooth shaded triangles/second with perspective correction at the cost of less than $1,000 USD.

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee
1997 SIGGRAPH/Eurographics Workshop
Copyright 1997 ACM 0-89791-961-0/97/8..$3.50

Despite the long history and recent furious race in 3D graphics hardware development, empirical studies on the characteristics of 3D graphics workloads whose results can be used to guide the architectural and implementation decisions have been curiously few in the literature. Descriptions of 3D graphics hardware architectures in earlier publications mostly focused on how graphics algorithms are translated into hardware, rather than on design tradeoffs that are optimized according to empirical workload statistics. The goal of this work is to fill this gap by reporting the result of a characterization study of static 3D graphics workloads based on an instrumented software renderer tested on a large set of complicated 3D models. By static graphics, we mean the positions and other properties of graphics objects in the 3D model do not change over time.

Published work in 3D graphics workload characterization is relatively scarce. Whelan's thesis [16] 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 slow by today's standard. Dunwoody [5] 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 software environment, rather than the detailed analysis of 3D workloads themselves. Cox and Hanrahan [3] 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 [2]. She studied nine highly complicated images from different periods of Pixar's animation productions by instrumenting proprietary rendering tools. She also focused most of the attention on depth complexity and visible surface measurements. Compared to previous work, the work reported in this paper is different in two important aspects First, the workload is based on 3D models from VRML (Virtual Reality Modeling Language) files freely available over the Internet. As a result, the results from this project can be easily verified and extended using the tool we developed. We believe this represents a giant step in 3D graphics workload study since most of the 3D model files used in previous studies are proprietary and thus not easily available. Secondly, the analysis of the 3D graphics workload characteristics presented in this work is specifically oriented towards architectural implications for graphics hardware implementations.

The rest of this paper is organized as follows. In Section 2, we briefly review the graphics pipeline for polygon rendering to facilitate the discussion of architectural implication of specific graphics statistics. In Section 3, the experiment methodology, including the instrumentation environment and the input 3D models, is described. Section 4 presents the detailed characteristics of the input 3D workloads, and their analysis from the viewpoint of graphics hardware/systems implementors. Section 5 concludes this paper with the major findings and an outline of the ongoing work along a similar line.

2 3D Polygonal Graphics Pipeline

The graphics primitives in a 3D model input file are organized as a tree. Rendering starts with a traversal through the tree of primitives. For each primitive, the renderer usually decomposes it into more easily manipulable units for rendering, such as polygons or triangles, if necessary. For 3D graphics rendering hardware, triangles are preferred. For example, our instrumented renderer divides a sphere into 26 triangles and 52 squares. The rest of the rendering process [6] consists of two distinct stages: *geometric transformation* and *rasterization*. The computation in the *geometric transformation* stage is mostly floating-point intensive, involving vector space operations such as matrix multiplication and inner products, and is also easily parallelizable. The *rasterization* stage, on the other hand, involves mostly integer arithmetic, involving simple additions and compares, but requires coordination of accesses to shared data structures during visibility computation and is thus more difficult to parallelize.

The first step in the *geometric transformation* stage is *view transformation*, in which the polygons and their normal vectors are first transformed from the world coordinate system to the eye coordinate system, taking into account the position and direction of the viewpoint. Then the polygons are transformed into the screen coordinate system, in which the positive X and Y axes extend to the right and up directions of the screen, while the positive Z axis points into the screen. Because of the screen size limitation, polygons that fall outside the visible frustum will be clipped. The portions of the polygons within the frustum form new polygons and may need to be decomposed into simpler rendering primitives. Perspective correction is performed in this step. The second step is *simple visibility culling*, which eliminates from further consideration the polygons that are exactly perpendicular to the screen, and those that form the *back* faces of opaque objects with respect to the current viewpoint. The final step is *shading*, in which the color of each vertex of each potentially visible polygon is computed according to the material property, the light source, and the assumed lighting/shading model. The color representation can be based on RGB (Red, Green, Blue) or YUV (Luminance and Chrominance). If the polygon is associated with a transparent object, a translucency parameter called the α value is also computed for subsequent compositing operations.

The *rasterization* stage itself comprises three steps. The *polygon processing* step decomposes a given polygon into a set of horizontal pixel segments called *spans*, and identifies the two end points of each span. For triangles, this step is trivial. For general polygons, a complete traversal through the polygon's edges to compute the X and Y coordinates of each pixel on the edges is required. The *edge walking* step computes the color, depth (or Z) value, and α value if applicable for each pixel on the edges. These values are derived from those associated with the edges' end points using linear interpolation along the Y direction. The *span processing* step calculates the color, depth (or Z) value, and α value if applicable for each pixel on each span, again using linear interpolation from the corresponding values of the end points of the span. There is a separate buffer of the same size as the screen called the *Z buffer*, which stores for each pixel the depth value of the portion of the polygon that contributes to that pixel, and its associated color and α values. During *span processing*, after the Z value of each pixel in the polygon is computed, it is compared with the current content of the Z buffer at the corresponding X-Y coordinate. If the new Z value is smaller, that means the newly computed pixel is closer to the eye than the one traversed earlier. Therefore, the current content of that Z buffer entry is replaced with the new pixel's Z value, color and α values. Otherwise, proceed with the next pixel in the polygon. This is assuming that objects are opaque, i.e., $\alpha = 1$. For transparent objects,

$\alpha < 1$, the colors are composited together to accumulate the final pixel color, according to their α values.

The above description of the 3D graphics pipeline implicitly assumes that the shading model is *Gouraud Shading*, the lighting model is *Diffuse Lighting*, and no texture mapping, which is a technique to use 2D images to emulate 3D geometric complexities. Of course, there are other more advanced graphics techniques that could further increase the realism of rendered images such as shadowing, anti-aliasing, and ray tracing. Most of these techniques will significantly increase the computational complexity of the rendering process. On the other hand, the described graphics pipeline is meant for hardware implementation and therefore doesn't include any sophisticated software optimizations, such as advanced visibility culling techniques, dynamically generated texture mapping, and object-space sorting [14]. The goal of this study is to measure the detailed characteristics of the set of tasks that a generic 3D graphics hardware pipeline has to perform so that empirical workload statistics can be used to guide the design of high-performance graphics engines.

To clarify how the set of tasks are affected by the geometric complexities of the input 3D model, Table 1 lists the basic geometric primitive with which each of the tasks is working. As can be seen, the computational requirements for different tasks depend on different levels of geometric details. For example, the amount of Shading computation is proportional to the total number of vertices in all the polygons, whereas the computational load due to Span Processing is determined by the number of pixels covered by the polygons.

Task	Primitive
View Transformation	per vertex of each polygon
Simple Visibility Culling	per polygon
Shading	per vertex of each polygon
Polygon Processing	per polygon
Edge Walking	per pixel on each edge
Span Processing	per pixel on each polygon

Table 1: *The computational requirement for each task in the 3D graphics rendering pipeline in terms of the basic geometric primitive.*

3 Experiment Methodology

The original motivations for our study of 3D graphics hardware is to improve the interpretation speed of VRML files during Internet accesses. Also, the fact that many VRML files are readily available over the net opens up, for the first time ever, the unique opportunity to examine highly sophisticated 3D models that were not previously possible. Therefore, this workload characterization study is entirely based on VRML and its renderer.

3.1 Instrumentation Environment

VRML [11], officially announced in May 1995, is a 3D graphics specification language based on Open Inventor [15], which itself is based on OpenGL [9] from Silicon Graphics, for defining 3D scenes, rather than textual/image descriptions, in documents that are linked through the World Wide Web technology. VRML allows definitions of geometry, text, grouping, levels of detail, transformation, material properties, rendering attributes, light sources, and camera positions. In addition, VRML provides a document linking mechanism similar to HTML. VRML 1.0 supports only *static* scene descriptions; dynamic or moving object behavior is included in VRML 2.0. This work only focuses on 3D models written in VRML 1.0.

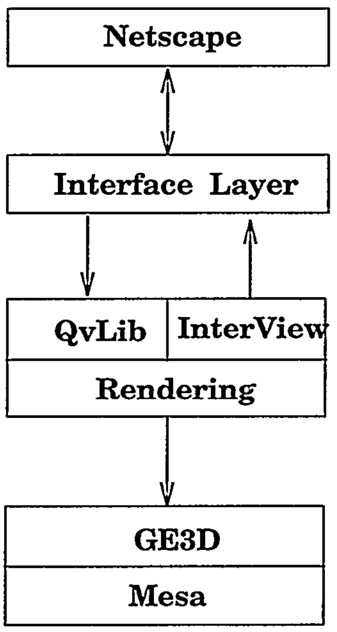

Figure 1: *The software architecture of VRweb. The instrumentation code is inserted into the Mesa library to measure various rendering statistics.*

The VRML-capable viewer we use is called *VRweb* [12], which is a VRML viewer that supports multiple Web protocols and whose software architecture is shown in Figure 1. We chose *VRweb* because its source code is freely available, which is mandatory for instrumentation. Within *VRweb*, the VRML files are first parsed by a freely available parser from SGI called *QvLib* [13], which translates VRML specifications into low-level OpenGL-like primitives, which are input to a public-domain 3D graphics rendering library called *Mesa* [10]. OpenGL is the most prominent 3D graphics API, a vendor-neutral standard endorsed by many major suppliers. Most of the OpenGL primitives are at a sufficiently low level that is amenable to direct hardware manipulation. In this study, the statistics collection code is inserted in the *Mesa* library to record relevant workload characteristics. As will be shown later, the instrumentation includes both geometric complexity of the input files and measurements that reflect the dynamics of the rendering pipeline.

3.2 3D Input Models

To test a wide variety of 3D models, this study focuses on four categories of VRML files, whose geometric characteristics in terms of the total file size, numbers of polygons and vertices are shown in Table 2. The first four files, Chamber, Hall, Lab, and Room are created from graphics modeling tools and libraries from Lightscape Technologies, and represent the high-quality polygon rendering category. They are all indoor scenes with the surfaces decomposed into very

Name	Texture Mapped	File Size	No. of Polygons	No. of Vertices
Chamber	NO	13.1MBytes	46,836	140,826
Hall	NO	9.6MBytes	41,202	123,924
Lab	NO	13.9MBytes	79,642	239,244
Room	NO	9.3MBytes	31,604	95,130
Atrium	NO	1.2MBytes	13,127	39,382
Exhibit	NO	557KBytes	8,345	25,306
Spiral	NO	2.2MBytes	22,600	67,800
Abbey	NO	8.5MBytes	58,818	229,671
Hilo	NO	2.6MBytes	34,203	102,927
Tee	NO	2.8MBytes	62,601	188,121
Asian	YES	38KBytes	545	2,180
Granmamt	YES	29KBytes	453	1,812
Launch1	YES	27KBytes	448	1,792
Steel	YES	14KBytes	222	888

Table 2: *Basic Characteristics of the 3D VRML files used in this study.*

small polygons to simulate the fine details of lighting effects. The next three files, Atrium, Exhibit, and Spiral, are generated by LightWork Design's radiosity processor, which puts similar emphasis on lighting effects as the previous four but uses less complex geometry. The next three, Abbey, Hilo, and Tee, are VRML files translated from architectural models that are originally in the AutoCAD format. The lighting and shading considerations are minimal but geometric accuracy is of major concern. The last four, Asian, Granmamt, Launch1, and Steel, are representative of 3D scenes that include texture maps. They are generated from VRML builders. The file size listed does not include the texture images themselves. A distinct characteristic of texture-mapped 3D scenes is that the geometric complexity is much less sophisticated compared to other non-texture-mapped scenes. This is because in texture-mapped models 2D texture images are used to emulate fine 3D geometric details, thus greatly simplifying the geometric description.

4 Workload Statistics and Analysis

The results of the characterization study are grouped into four subsections. In the first subsection, the effectiveness of simple culling techniques used in *Mesa* is evaluated in terms of the percentage of polygons eliminated from further consideration. Among the polygons that need to be rasterized, the input statistics that affect the rasterization performance such as the average span and edge length are also presented. Secondly, the depth complexity of the workloads is examined from three different perspectives. Thirdly, the compressibility of computer graphics generated bitmaps is analyzed for a simple compression scheme that is based on 3D rendering. Finally, the impact of viewpoint changes on the various workload statistics is discussed.

4.1 Effectiveness of Culling

Table 3 shows the effectiveness of *Mesa*'s culling techniques in reducing the number of polygons that are actually propagated to the rasterization stage, called *rasterized* polygons. *Clipped* polygons are those that completely fall outside of the view volume, according to a simple check based on world coordinates. *Back-faced* polygons are those that are hidden behind opaque objects from the current viewpoint, according to a test on their transformed normal vectors. *Zero-sized* polygons are non-clipped and non-back-faced polygons those whose effective visible area is too small to make any contribution. Usually these polygons are too far away from the viewer position to contribute to the rendered image. Zero-sized polygons can only be identified after their vertex coordinates are transformed. In *Mesa*, the vertex coordinates of zero-sized and rasterized polygons are always transformed during the view transformation step. However, only rasterized polygons are shaded, i.e., the colors and α values are actually computed. From Table 3, decoupling of shading from geometric transformation can indeed save a significant amount of computation, ranging from 26.54% (*Exhibit*) to 89.64% (*Steel*). Zero-sized and rasterized polygons are collectively referred to as *rendered* polygons hereinafter. Some 3D models include texture mapped polygons, the percentages of which that are actually rendered with respect to the total number of polygons in the model are indicated by the last column of Table 3.

Among the rendered polygons, the workload parameters that affect rasterization performance are listed in Table 4. The length of the polygon edge is measured in terms of the effective extent along the Y direction, which affects the computation requirements during polygon processing and edge walking. The span width is measured along the X direction, and it determines the computational requirement for span processing, including the number of Z buffer access and comparison. The numbers of edges and spans for texture-mapped models are much smaller than non-texture-mapped models because the total numbers of polygons are inherently smaller in the former. Accordingly, the average edge and span sizes for texture-mapped models are larger than non-texture-mapped models. From hardware implementation standpoint, longer edge and span sizes are preferred since the fixed start-up cost for processing each edge and span can be amortized over a larger number of steps. To further understand the distribution of span length, Figure 2(a) shows the histogram of span size for the non-texture-mapped 3D models. A great majority of spans have a size smaller than 5, indicating that spans are usually too small to allow for efficient pipelining of span processing.

Name	Clipped	Back-Faced	Zero-Sized	Rasterized	Textured
Chamber	18.04%	12.76%	13.47%	55.73%	0%
Hall	15.51%	36.55%	13.62%	34.32%	0%
Lab	37.02%	17.73%	12.55%	32.70%	0%
Room	33.53%	17.72%	7.86%	40.84%	0%
Atrium	53.82%	17.78%	1.20%	27.20%	0%
Exhibit	11.68%	7.93%	6.94%	73.46%	0%
Spiral	74.84%	5.04%	0.80%	19.32%	0%
Abbey	5.66%	0.30%	40.42%	53.61%	0%
Hilo	2.70%	31.08%	24.91%	41.30%	0%
Tee	0.49%	0.00%	53.97%	45.53%	0%
Asian	41.10%	17.80%	6.24%	34.86%	34.86%
Granmamt	10.82%	48.79%	1.32%	39.07%	25.61%
Launch1	2.90%	39.96%	4.46%	52.68%	20.76%
Steel	76.13%	13.06%	0.45%	10.36%	10.36%

Table 3: *The percentages of polygons that are eliminated due to culling techniques used in* Mesa.

Name	No. of Edges	Avg Edge Length	No. of Spans	Avg Span Width
Chamber	73,856	3.89	101,333	4.95
Hall	41,698	21.90	231,421	2.90
Lab	82,909	5.43	124,156	3.77
Room	35,658	6.64	74,635	5.38
Atrium	9,238	16.67	67,315	10.15
Exhibit	16,136	5.03	34,342	8.53
Spiral	12,368	9.88	54,043	8.56
Abbey	137,090	3.84	183,214	3.99
Hilo	46,956	8.21	117,160	10.05
Tee	108,526	9.77	176,980	6.59
Asian	587	18.41	5,387	76.60
Granmamt	434	28.79	6,224	91.19
Launch1	792	26.02	9,790	60.36
Steel	86	182.30	7,599	66.59

Table 4: *The workload parameters that affect the performance of the rasterization stage, i.e., polygon processing, edge walking, and span processing.*

Name	Polygon Depth	Span Depth	Pixel Depth
Chamber	1.45	1.56	1.51
Hall	2.68	1.97	2.01
Lab	1.37	1.39	1.41
Room	1.41	1.33	1.21
Atrium	1.88	1.74	2.05
Exhibit	1.46	1.37	1.27
Spiral	1.30	1.30	1.39
Abbey	3.07	3.51	3.71
Hilo	3.35	5.70	4.92
Tee	6.29	7.06	5.68
Asian	1.28	1.52	1.33
Granmamt	3.22	2.29	1.81
Launch1	1.43	1.45	1.85
Steel	1.15	1.21	1.52

Table 5: *The depth complexity of the 3D models as measured from the polygon, span, and pixel standpoints. The span and pixel depth complexities are only contributed by the polygons that pass Mesa's culling techniques.*

Name	Compressible Spans	Uncompressible Spans	Compression Efficiency
Chamber	45.21%	54.79%	38.22%
Hall	27.65%	72.35%	57.79%
Lab	46.41%	53.59%	52.49%
Room	55.22%	44.78%	34.54%
Atrium	62.12%	37.88%	25.16%
Exhibit	62.03%	37.97%	16.78%
Spiral	65.78%	34.22%	28.34%
Abbey	38.65%	61.35%	32.43%
Hilo	55.58%	44.42%	13.81%
Tee	30.55%	69.45%	24.78%
Asian	15.50%	84.50%	54.07%
Granmamt	18.19%	81.81%	90.34%
Launch1	28.25%	71.75%	73.93%
Steel	0.02%	99.98%	101.24%

Table 6: *The percentages of visible spans that are compressible and uncompressible, and the overall compression efficiency, which are the smaller the better.*

4.2 Depth Complexity

An important optimization technique to improve the performance of rasterization is to avoid processing the polygons that are hidden from others with respect to the current viewpoint. Two separate questions need to be addressed that are related to the feasibility of this approach. First, what is the percentage of polygons rasterized that are *visible* polygons, those that contribute to at least one pixel in the final image? If the percentage is low, it means that there is ample room for optimization through visibility pre-processing so that completely invisible polygons can be precluded from further manipulation early on in the geometric transformation stage. The ratio between the number of rendered polygons to that of visible polygons is called *polygon depth complexity*. Secondly, what is the average number of polygons that overlap on a given pixel? The notion of *pixel depth complexity* is usually defined in terms of the number of polygons in the model that cover a given pixel for a specific viewpoint. In this study we only measure the depth complexity contributed by those polygons that pass through *Mesa*'s simple culling techniques. During the span processing step, two operations are performed: depth sorting through Z buffering and pixel coloring. Although depth sorting needs to be performed for every polygon pixel, pixel coloring can be deferred until it is known that the corresponding polygon pixel indeed contributes to the final image. This idea is called *lazy shading* [4] [8], and is only applicable for opaque polygons. The larger the pixel depth complexity, the more overhead saved using lazy shading, especially when the pixel coloring operation involves expensive computation such as anti-aliasing and texture mapping. Table 5 lists the depth complexity of the examined 3D models from the polygon, span, and pixel standpoints. The *span depth complexity* is defined as the ratio between the number of rendered spans and the number of visible rendered spans, those that actually contribute to the final image. Although not immediately useful, *span depth complexity* indicates the potential performance improvement of visibility pre-processing if it is performed at the span rather than polygon level. All the depth complexity measures exclude the background color. In general, these three depth complexity measurements are roughly but not strictly correlated.

To further examine the distribution of pixel depth and the effectiveness of lazy shading, Figure 2(b) shows the histogram of the pixel depth in the ten non-texture-mapped models. The first several models are mainly indoor scenes and therefore have less depth complexity. On the other hand, the remaining three are AutoCAD models, which tend to cover a larger physical scope and thus exhibit higher depth complexity. Transparent polygons have an α value smaller than one. Among the 14 models examined, only two of them contain transparent polygons, *Abbey* and *Tee*, and the percentages of transparent polygons among all polygons are 0.36% and 2.87%. Therefore, lazy shading can be applied to most of the models quite effectively.

4.3 Compressibility

Because of the demanding computational requirements of high-quality 3D graphics processing, high-performance rendering servers are shared among a set of client machines on which end users develop graphics applications. When the rendering process is completed, bitmaps are transferred over the network to the clients for display. To conserve the network bandwidth, bitmaps should be compressed, preferably in a lossless fashion. Rather than apply a generic lossless compression algorithm to this problem, we make the following observation: The process with which the rendering servers transform 3D models into 2D bitmaps can be considered as a decompression process. Therefore, the best way to compress the 2D bitmaps is *to carefully divide the 3D rendering pipeline between the server and the client so that the graphics representation that is transferred over the network is as compact as possible while the remaining processing at the client side is as simple as it can.* With this approach, there is no need to explicitly compress at the server end, and decompression at the client side simply means finishing the rendering pipeline.

One possible division of the graphics pipeline is to ship the visible spans that are determined at the end of span processing to the clients, which then expand them into pixel colors. Each span is represented as the R, G, B values of its starting end point, the R, G, B deltas for expanding the rest of the span, its span length, and the length of the following uncompressible span, if any. The spans are concatenated to cover the entire screen space so that there is no need for explicit X and Y coordinates for each span, assuming that the spatial dimensions of the screen is known. Because each span presentation costs 8 bytes, only spans of length more

View	Percentage of Rasterized Polygons	Pixel Depth	Compression Efficiency
1	45.53%	5.68	24.78%
2	43.93%	7.48	33.03%
3	40.55%	5.90	13.82%
4	37.09%	6.15	7.39%
5	40.81%	5.30	11.32%
6	44.35%	5.84	19.64%
7	47.30%	6.83	21.68%
8	39.54%	8.09	23.16%
9	38.55%	9.99	38.81%

Table 7: *The percentage of rasterized polygons among all polygons in the model, the pixel depth, and the compression efficiency measured from different viewpoints for the 3D model,* Tee.

than two are compressed and are called *compressible spans.* Those with length less than three are represented explicitly. Texture-mapped spans are also represented explicitly because the clients don't have the texture images to expand the spans. Similarly, transparent spans are also represented explicitly. Spans shorter than three pixels, texture-mapped and transparent spans are called *uncompressible spans.* Table 6 shows the percentages of compressible and uncompressible visible spans and the overall compression efficiency, which is defined as the ratio between the sizes of the compressed and uncompressed representations. This table does include spans from the background color. As expected, non-texture-mapped models exhibit a higher compression gain, ranging from 1.73 to 7.24, than texture-mapped ones, where the compression gain is smaller than two and mainly results from background spans. *Steel* is particularly bad because almost all its polygons are texture mapped and thus the proposed compression scheme can not apply at all. Note that it is possible to apply generic lossless compression techniques such as *gzip* to achieve further compression gains.

4.4 Effect of Viewpoints

Given a 3D model, rendering from different viewpoints may entail different computational requirements. Table 7 shows three visibility-related statistics for rendering computations from different viewpoints of the 3D model, *Tee.* Although there are small differences in the ratios between the number of polygons that are passed to the rasterization stage and the total number of polygons in the model, the pixel depth and compression efficiency vary significantly. Moreover, there doesn't appear to have any correlation between rasterized polygon percentages and pixel depth, which is not surprising since the pixel depth is measured among the rasterized polygons only, rather than all the polygons in the model. Similarly, compression efficiency has more to do with visible span length than with depth complexity.

5 Conclusion

Despite the recent heated interest in 3D graphics hardware such as Microsoft's Talisman initiative [14], Intel's Accelerated Graphics Port [1] system architecture, and the proliferation of 3D programming APIs and graphics cards, systematic studies on the 3D graphics workloads and their architectural implications are conspicuously lacking. The work reported in this paper takes the first step to fill this gap by studying the detailed rendering characteristics of a wide variety of static 3D graphics models with different geometric and texture properties. Wherever appropriate, the implications of these statistics on the graphics pipeline performance are analyzed and presented.

There are several directions that we are currently taking to continue the comprehensive 3D graphics workload characterization study. First, we would like to study the 3D workload in which the relative positions between the viewpoint and graphics objects are dynamically changing, for example, in a computer game or virtual world simulation environment. In this case, the performance goal of graphics hardware design is shifted from photo-realism to cinema-realism. Algorithms that could quickly approximate exact rendered images play a critical role in this type of applications. Also, we would like to continue this study by re-examining the statistics when advanced polygon rendering techniques are included, in particular, anti-aliasing and variants of texture mapping such as bump mapping. Characteristics of other rendering models such as volume rendering, radiosity, and ray tracing will also be included as part of the continuing effort of 3D graphics workload characterization.

Acknowledgement

This research is supported by an NSF Career Award MIP9502067 and a contract 95F138600000 from Community Management Staff's Massive Digital Data System Program.

Reference

[1] *Accelerated Graphics Port Interface Specification*, Revision 1.0, http://www.intel.com/pc-supp/ platform/ agfxport/agpl August 1996.

[2] Blau, R, *Performance evaluation for computer image synthesis systems.* Thesis (Ph.D.). of University of California, Berkeley, UCB/CSD 93/736, 1992.

[3] Cox, M.; Hanrahan, P., "Depth complexity in object-parallel graphic architectures," Princeton University. Department of Computer Science. CS-TR-382-92, 1992.

[4] Chiueh, T., "Heresy: A Virtual Image-Space 3D Rasterization Architecture," 1997 ACM Siggraphs/Eurographics Workshop on Graphics Hardware.

[5] Dunwoody, J.C.; Linton, M.A., "Tracing interactive 3D graphics programs," Computer Graphics (March 1990) vol.24, no.2, p. 155-63.

[6] Foley, J.D.; van Dam, A.; Feiner, S.; Hughes, J., *Computer Graphics: Principles and Practice*, Addison-Wesley Pub. Co., Reading, Massachusetts, second edition, 1990.

[7] MacIntyre, B., "PC 3D Graphics Accelerator FAQ," http://www.cs.columbia.edu/ bm/ 3dcards/3d-cards1.html,

March 1996.

[8] Molnar, S.; Eyles, J.; Poulton, J, "PixelFlow: High-Speed Rendering Using Image Composition," *Proceedings of the 19th SIGGRAPH Conference*, pp. 231-240, Chicago, Illinois, July 1992.

[9] Neider, J.; Davis, T.; Woo, M., *OpenGL Programming Guide*, Addison-Wesley, 1993.

[10] Paul, B., "The Mesa-3D graphics library," http://gopher.ssec.wisc.edu/ brianp/Mesa.html.

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

[12] Pichler, M.; Orasche, G.; Andrews, K.; Grossman, E.; McCahill, M., "VRweb: a multi-system VRML viewer," Proceedings of the first annual symposium on the Virtual Reality Modeling Language (VRML '95), San Diego, California, December, 1995.

[13] Strauss, P.; Bell, G., "The VRML Programming Library," http://vag.vrml.org/www-vrml/vrml.tech/qv.html.

[14] Torborg, J.; Kajiya, J., "Talisman: Commodity Real-time 3D Graphics for the PC," http://www.research.microsoft.com/SIGGRAPH96/Talisman/, also SIGGRAPH96, August 1996.

[15] Wernecke, J., *The Inventor Mentor*, Addison-Wesley, 1994.

[16] Whelan, D. S., *A multiprocessor architecture for real-time computer animation*, 5200:TR:85, California Institute of Technology. Computer Science Department, 1985.