Tracing Interactive 3D Graphics Programs

J. Craig Dunwoody and Mark A. Linton
Center for Integrated Systems
Stanford University
Stanford, California 94305

Abstract

The two goals of graphics performance analysis are to characterize application workloads and to understand how systems perform under these workloads. We have developed a set of tools to help achieve these goals. TGEN is a tracing program that intercepts graphics library calls from an application program and records them in a file. TPROF is a profiler that interprets a trace file and computes workload statistics. TBENCH is a portable performance measurement tool that executes a trace file on a graphics system and measures the resulting update time. In this paper, we describe these tools and give examples of their use.

1 Introduction

The most accurate way to measure the performance of an interactive 3D graphics system is to apply real workloads. Improving the performance of graphics applications and systems requires workload analysis. The tasks of measuring and improving performance can be made easier through the development of specialized tools.

Application performance can be measured without special tools; one simply executes a program and measures display update times directly. This approach can be quite inconvenient, however, for the following reasons:

1. Building a program for the system under test requires access to source code, which is often difficult to obtain.
2. Because most existing programs use proprietary graphics libraries, a significant porting effort will usually be necessary.
3. Most interactive programs are not equipped for benchmarking. Such use typically requires program modifications or an external scripting facility, which may not work reliably.
4. The data files and user expertise necessary to operate a program in a manner representative of typical usage may be unavailable.
5. The system under test may exist only as a simulation, making interaction with programs difficult or impossible.

Similarly, a useful amount of workload analysis can be done by tracking graphics-library calls using a general-purpose profiler. More sophisticated analysis, however, requires tools that understand what is going on inside the graphics system.

We have developed a set of tools that is useful for both performance measurement and workload analysis. Our approach, illustrated in Figure 1, is to generate a trace of the graphics library function calls that a program makes during execution. We can use this trace as an application-specific benchmark by executing it on other graphics systems and measuring update times. We can also gain insight into the program's behavior by analyzing the trace file.

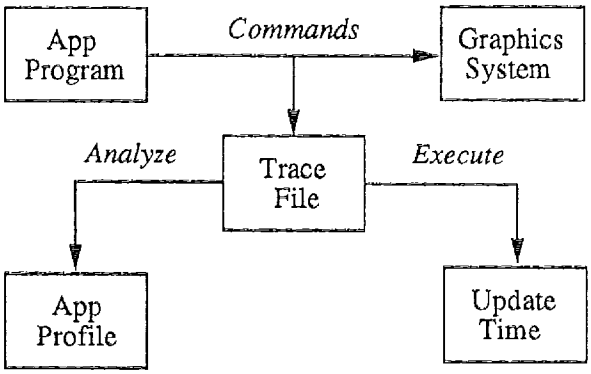


Figure 1: Trace-based profiling and benchmarking

In order to represent traces in a portable way, we designed a graphics interface called GR. Each trace is a binary or textual encoding of a sequence of GR calls. The GR interface is simple and general enough to be implemented efficiently on top of existing graphics libraries.

In this paper, we discuss the design of GR and describe how a trace is generated. We then present our profiling and benchmarking techniques, along with the results that we obtained by applying them to several application programs.

2 The GR Interface

The functionality of a general-purpose interactive 3D graphics system is defined by the application-program interfaces (APIs) that

This research has been supported by the Quantum project through a gift from Digital Equipment Corporation.

it provides. Most of these APIs include at least the following features:

- Primitives: polypoint, polyline, polygon, triangle strip, text, clear viewport
- Transformations: 4x4 floating-point matrix stack
- Shading: coloring (flat/smooth), multiple-source lighting (flat/smooth), depth cueing
- Hiding: backface culling, Z-buffer
- Display: double buffering, multiple overlapping windows

Following Akeley[1], we can view the interactive-graphics update process as a six-stage pipeline: input handling, database modification, database traversal, transformation, rasterization, and display. As shown in Figure 2, the API may be placed either between the modification and traversal stages (a "structure" interface) or between the traversal and transformation stages (an "immediate" interface).

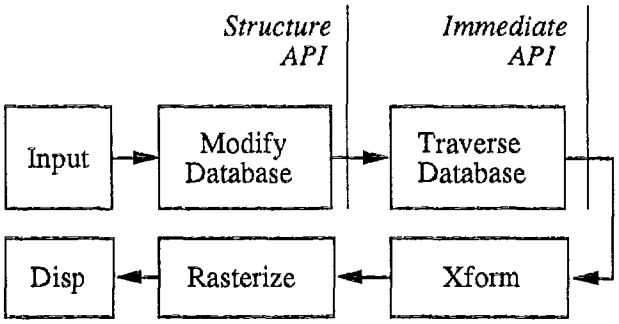


Figure 2: Update pipeline

Most systems provide an immediate interface with one or more structure interfaces layered on top. We decided to generate traces at the level of the immediate interface, for two reasons. First, the immediate interface provides a common tracing point for all applications, whether or not a structure API is used. Second, instructions at the immediate interface are simpler, and it is therefore easier to analyze traces and interpret them efficiently on top of existing graphics libraries.

No immediate interface has achieved widespread use across multiple vendors. APIs with immediate capability include GL from Silicon Graphics[7], Starbase from Hewlett-Packard[3], On-Ramp from Tektronix[8], and Doré[4] and XFDI[2] from Stardent. The PEX[6] standard shows promise, but implementations are not yet widely available.

Because there was no standard immediate interface to serve as the basis for our portable trace format, we chose to design our own simplified immediate interface called GR and implement translators between GR and existing interfaces. For recording traces, we have implemented a GL-to-GR translator. For interpreting traces, we have implemented GR-to-GL and GR-to-Starbase translators.

2.1 Design Principles

GR is a RISC graphics instruction set. We based the GR design on the following principles:

1. Include only functionality that is necessary to represent efficiently the behavior of existing graphics programs. Convenience functions can be layered on top.
2. Make each function do a very small amount of work. This promotes extensibility and interpretation efficiency, and it opens up the possibility of applying an optimizing compiler to trace files.
3. Keep the amount of state that must be maintained by an interpreter to a minimum.
4. Make the interface resolution-independent. Screen coordinates and color values should be given as normalized floating-point numbers where it is possible to do so without compromising interpretation efficiency.
5. Avoid unnecessary device dependencies, but do not limit functionality to the least common denominator. Every graphics API implements some subset of GR features. Each GR interpreter reports to its client on the use of unimplemented features; the client can then decide on the appropriate action.
6. Offer enough flexibility in the interface to insure that traces can be interpreted efficiently on a wide range of graphics systems. For a given GR feature, there are variations in the interface provided by existing graphics libraries. Most of these variations can be hidden by a GR interpreter without significant performance impact. An important exception is the formatting of vertex data for primitives. The most efficient format for this information varies among existing graphics libraries. GR provides a flexible scheme for specifying vertex data format. TBENCH automatically filters vertex data into the most efficient format before executing a trace.
7. To make tracing simpler, pass only primitive data types (integer, float, string) across the interface, and refer to resources using table indices rather than pointers to dynamically allocated objects.

2.2 Function Groups

Table 1 lists the five groups of basic GR functions: display, hiding, transformation, shading, and primitives. The display group supports the manipulation of multiple independent, overlapping, double-buffered windows. There is a fixed table of windows, all of which are initially unmapped. All GR state is per window, except the current window index.

The hiding group supports backface culling and Z-buffering. The Z-buffer may be cleared independently or simultaneously with the color buffers. There is a performance advantage to simultaneous clearing on some systems.

The transformation group supports operations on two 4x4 matrices: the modeling matrix, which transforms user coordinates into the world space where lighting calculations can be done, and the projection matrix, which transforms world coordinates into normalized screen coordinates.

The shading group supports five types of shading: colored-flat (per-primitive color), colored-smooth (per-vertex color), lighted-flat (per-primitive normal), lighted-smooth (per-vertex normal), and lighted-colored-smooth (per-vertex normal with per-vertex color modulating surface reflectance). When lighting is enabled, shade is determined by the position and color of the light sources, the position and surface reflectance of the primitives, and the position of the viewer. The viewer and each light source may be either directional (infinite) or positional (local). Lighting calculations are more compute-intensive when viewer and light sources are positional.

Group	Function	Description
Disp	winindx	Select window
	winpos	Set window position
	winsiz	Set window size
	winmap	Map window to display
	winclip	Set clip rectangle
	winport	Set viewport
	winclr	Clear viewport
	coldst	Enable color buffers
	colswp	Swap color buffers
Hide	polycull	Enable backface cull
	depthdst	Enable Z-buffer
Xform	mtxindx	Select matrix (model/proj)
	mtxld	Load matrix
	mtxmul	Multiply matrix
	mtxpop	Pop matrix stack
	mtxpush	Push matrix stack
Shade	col	Set flat-shade color
	nml	Set flat-shade normal
	lteon	Enable lighting
	lteamb	Set ambient light color
	lteatt	Set light attenuation
	lteloc	Set viewer position
	ltmamb	Set ambient reflectance
	ltmdiff	Set diffuse reflectance
	ltmspec	Set specular reflectance
	ltmspecexp	Set specular exponent
	ltson	Enable light source
	ltsemit	Set light source color
	ltspos	Set light source position
Prim	vtxbgn	Start vertex definition
	vtxadd	Add position/color/normal
	vtxend	Finish vertex definition
	vtx	Send array of vertices
	pntbgn	Start polypoint
	pntnxt	Finish/start polypoint
	pntend	Finish polypoint
	linbgn	Start polyline
	linnxt	Finish/start polyline
	linend	Finish polyline
	polybgn	Start polygon
	polynxt	Finish/start polygon
	polyend	Finish polygon
	tribgn	Start tri-strip
	trinxt	Finish/start tri-strip
	triend	Finish tri-strip

Table 1: Basic GR functions

The primitives group supports polypoints, polylines, polygons, and triangle strips. Primitives are generated by a sequence of the form

```
<prim>bgn(); vtx(); vtx(); <prim>end();
```

The vertices can be provided collectively in a single call or individually in multiple calls, depending on the needs of the application. For maximum efficiency, TBENCH always provides all vertices in a single call. Vertex data format (position with optional color and normal) is defined procedurally by a sequence of the form

```
vtxbgn(); vtxadd(); vtxadd(); vtxend();
```

This scheme has two benefits: traces can be filtered into the most efficient format for a given system, and new vertex data components such as texture coordinates can be supported in the future without changing the interface.

The "next" calls encode the commonly occurring end-begin sequence. We define a *run* as begin-end sequence containing an arbitrary number of next and vertex calls and no state-changing commands.

3 Trace Generation

TGEN is a shell that executes an interactive graphics program and writes trace files. We designed TGEN to meet the following requirements:

1. The traces must consist of a sequence of GR calls.
2. TGEN must provide access to a large body of interactive 3D graphics programs.
3. TGEN must be convenient to develop and install. It must be an ordinary user program and not require any changes to system software.
4. TGEN must be convenient to use. It must not require the user to recompile or relink the program being traced, because the program may be a proprietary package for which source code is not available.
5. TGEN must operate transparently. It must not affect the behavior of the program being traced.
6. The traces must be as compact as possible, for ease of storage, processing, and transmission.

Since there are no programs that directly issue GR calls, Requirements 1 and 2 mean that TGEN must incorporate a translator that converts calls to an existing graphics library into GR calls. For the initial implementation of TGEN, we chose Silicon Graphics GL.

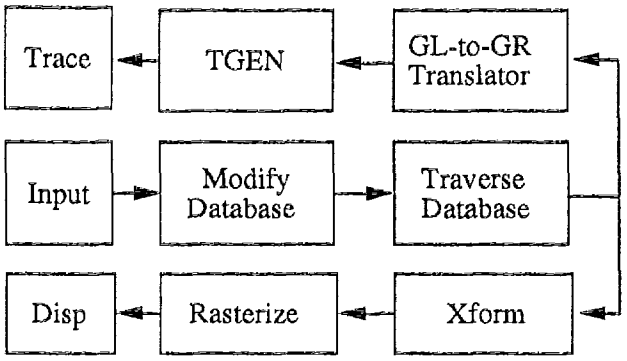


Figure 3: Trace generation

Calls into GL go through a shared-library branch table containing pointers to GL functions. This made it possible for TGEN to meet Requirements 3 and 4. When TGEN starts a graphics program, it replaces all of these pointers with a pointer to a single tracing function. This tracing function converts each incoming GL call into the GR equivalent, records the GR call in a trace file, and then calls the normal GL function.

If TGEN recorded traces continuously, it would be impossible to meet Requirements 5 and 6. Some graphics systems can render hundreds of thousands of primitives per second. At such rates, continuous tracing would require tens of megabytes per second of disk bandwidth, and the size of traces would be unmanageable.

Because continuous tracing is impractical, the TGEN tracing function is normally disabled. When the user sees a scene he would like to capture, he presses the PrintScreen key and TGEN generates a trace file containing a preamble and a single frame. The preamble is the sequence of GR calls necessary to bring a GR interpreter from its default state to the state that existed at the beginning of the frame. The frame itself can be generated in one of two ways:

1. TGEN sends a REDRAW event to each of the application program's windows and records all graphics-library calls until the application again asks for input. This method is useful because it captures the complete contents of all of the application's windows, and it works even if the application does not use double buffering.
2. TGEN starts recording graphics calls when the application performs a buffer swap, and it stops recording when the application performs another buffer swap in the same window. Although it only works for double-buffered applications, this method is more useful for performance analysis because it captures just the graphics calls made by the application during animation. To improve performance, many applications render parts of the display such as menus during redraws but not during animation.

In either case, because the TGEN tracing function is enabled only while a frame is being recorded, the program runs at nearly full speed. By pressing the PrintScreen key at several points in the execution of the program, the user can generate a set of representative single-frame trace files. Since most graphics programs exhibit a great deal of frame-to-frame coherence, this sampling technique is not much less accurate than continuous tracing, and it is much more efficient.

TGEN generates trace files in a compact binary format. Another tool called TFILT can convert traces between binary and an equivalent ASCII format that is convenient for editing. All of our tools accept traces in either format.

4 Example Traces

We used TGEN to create 28 traces from various programs running on a Silicon Graphics 4D/70GT. We assigned one of the twelve types defined in Table 2 to each trace, based on the dominant primitive and shading mode. A brief description of each trace follows.

Type 1 (polypoint colored-smooth)

VOXEL: a volume rendering of a cylinder head (data provided by Vital Images).

Type 2 (polyline colored-flat)

VASE0: a line rendering of a vase from a surface modeling program.

CFD: flow lines from a fluid jet hitting a flat plate.

CAR0: a line rendering of an automobile (data provided by Chrysler Corporation).

Type	Primitive	Shading
1	Polypoint	Colored-Smooth
2	Polyline	Colored-Flat
3	Polyline	Colored-Smooth
4	Polyline	Lighted-Smooth
5	Polygon	Colored-Flat
6	Polygon	Colored-Smooth
7	Polygon	Lighted-Flat
8	Polygon	Lighted-Smooth
9	Polygon	Lighted-Colored-Smooth
10	Tri-strip	Colored-Smooth
11	Tri-strip	Lighted-Smooth
12	Tri-strip	Lighted-Colored-Smooth

Table 2: Trace types

Type 3 (polyline colored-smooth)

ROD0: a line rendering of a connecting rod with stress values mapped to colors (data provided by Cray Research).

Type 4 (polyline lighted-smooth)

STICK0: a line rendering of a candlestick.

Type 5 (polygon colored-flat)

COPTER: A helicopter (data provided by SimGraphics Engineering).

Type 6 (polygon colored-smooth)

BARC: a radiosity rendering of a building interior.

Type 7 (polygon lighted-flat)

RAY: a simulation of a ray tracer.

OILRES: subsurface pressures in an oil field.

Type 8 (polygon lighted-smooth)

BALLS: a simulation of bouncing balls with elastic collisions.

LIGHTS: light sources bouncing in a cube.

LATHE: a simulation of metal cutting on a lathe.

VASE1: a surface-rendered version of VASE0.

STICK1: a surface-rendered version of STICK0.

BARS: stresses in a series of metal bars.

FLIGHT: an out-the-window view from a flight simulator.

VORO: an view from a program for exploring Voronoi sets.

Type 9 (polygon lighted-colored-smooth)

ROD1: a surface-rendered version of ROD0.

Type 10 (tri-strip colored-smooth)

SLICE: two cut planes through a 3D raster data set.

WARP: a sampled image converted into a triangle mesh and warped.

WELL: seismic data mapped around a series of wells.

Type 11 (tri-strip lighted-smooth)

IDEAS: a frame from a flying-logo animation.

SHELLS: isovalue surfaces.

GRID: layers of map data (data provided by Radian Corporation).

ROTOR: a turbine impeller (data provided by CISI-GRAPH).

CAR1: a surface-rendered version of CAR0.

Type 12 (tri-strip lighted-colored-smooth)

PIC3D: 2D seismic image data viewed as a height field.

All of the traces except VOXEL are from double-buffered programs and were generated with the buffer-swap technique. The VOXEL trace was generated from a two-window program using the REDRAW technique, but the extra graphics calls resulting from the full redraw are not significant. Plate 1 contains the images generated by the traces.

5 Trace-based Profiling

In order to gain a better understanding of the workload presented to a graphics system by real application programs, we wrote a dynamic profiler called TPROF. As shown in Figure 4, TPROF feeds a trace to a GR profiler module, which tracks state changes and counts primitives. The profiler forwards GR calls to a variant of the GR-to-GL interpreter that operates the graphics pipeline in a feedback mode, returning primitives after transformation, clipping, and culling. The profiler uses this information to compute statistics on what is actually sent to the rasterizer.

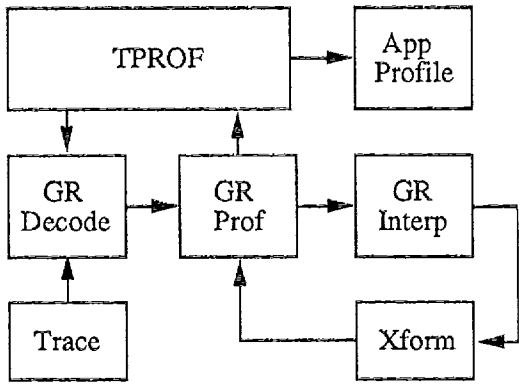


Figure 4: Trace-based profiling

The data generated by TPROF can be divided into three categories: global, geometry-related, and pixel-related. Table 3 presents the global data for the example traces, which consists of the following items:

- The total number of primitives.
- The fraction of primitives passed to the rasterizer after clipping and backface culling.
- The settings of the hidden-surface flags (backface-cull and Z-buffer). The value of a flag is recorded as true if it was turned on at any point in the frame; this can be misleading if it was only enabled for a small fraction of the primitives.
- The number of directional and positional light sources used.
- The number of windows and the number of times the current window was changed. Since each window change entails a potentially expensive context switch, unnecessary changes should be avoided.

Type	Trace	Prim Total	Pass Pct	Cull/ Zbuf	Ldir/ Lpos	Wcnt/ Wchg
1	VOXEL	1370	100	0/0	0/0	2/3
2	VASE0	785	100	0/0	0/0	4/0
	CFD	7483	100	1/1	0/1	2/0
	CAR0	35700	100	0/1	0/0	1/0
3	ROD0	830	100	0/1	0/0	1/0
4	STICK0	4203	100	0/1	3/0	1/0
5	COPTER	224	93	0/1	0/0	1/0
6	BARC	2677	76	0/1	0/0	1/0
7	RAY	704	100	0/1	0/1	3/2
	OILRES	4801	69	1/1	1/0	7/0
8	BALLS	414	100	0/1	1/0	1/0
	LIGHTS	457	100	0/1	0/3	1/0
	LATHE	224	100	0/1	1/0	1/0
	VASE1	758	100	0/1	1/0	4/0
	STICK1	2102	100	0/1	3/0	1/0
	BARS	2879	100	0/1	4/0	1/0
	FLIGHT	1673	65	1/1	1/0	1/0
	VORO	6823	100	0/1	2/0	1/0
9	ROD1	830	100	0/1	1/0	1/0
10	SLICE	154	100	0/1	0/0	4/0
	WARP	326	100	0/1	0/0	1/0
	WELL	2518	100	0/1	0/0	4/2
11	IDEAS	355	97	0/1	2/0	1/0
	SHELLS	1617	100	0/1	1/0	4/0
	GRID	452	100	1/1	1/0	3/0
	ROTOR	992	100	0/1	2/0	1/0
	CAR1	13675	100	0/0	1/0	1/0
12	PIC3D	137	95	0/0	1/0	3/7

Table 3: Global profile data

The transformation-time cost of backface culling must be weighed against the rasterization-time savings. Most of the traces relied exclusively on the Z-buffer for hidden-surface elimination. In two of the four cases where culling was used, no benefit was realized. In the OILRES trace the rasterizer load was reduced considerably, while in the FLIGHT trace the removal of primitives was mostly due to front-plane clipping rather than culling.

The majority of the multi-window programs used only a single window during animation. The PIC3D program would perform better on some systems if the number of unnecessary window changes was reduced.

Table 3 presents the geometry-related data for the example traces, which consists of the following items:

- The number of vertices transformed.
- The average number of vertices in lines, polygons, and triangle strips.
- The average length of runs of lines, polygons, and triangle strips.

In most cases, primitives of the dominant type were sent in large batches. These traces should execute efficiently even on systems for which state changes are slow. In three of the flat-shaded traces

Type	Trace	Vtx Total	LinVtx/ LinRun	PolyVtx/ PolyRun	TriVtx/ TriRun
1	VOXEL	267760	13/1	7/1	-
2	VASE0	2351	3/784	-	-
	CFD	15521	2/2	5/5	5/8
	CAR0	139714	4/637	-	-
3	ROD0	3313	4/828	-	-
4	STICK0	8403	2/4201	-	-
5	COPTER	814	-	4/1	-
6	BARC	10704	-	4/2676	-
7	RAY	2940	6/1	4/1	-
	OILRES	19200	4/1920	4/1	-
8	BALLS	1224	2/12	3/80	-
	LIGHTS	2160	-	4/144	18/8
	LATHE	1140	-	5/1	-
	VASE1	3024	-	4/756	-
	STICK1	8401	-	1/2100	-
	BARS	11524	4/1	4/99	-
	FLIGHT	5802	2/6	4/26	-
	VORO	26271	-	4/35	-
9	ROD1	3313	-	4/828	-
10	SLICE	9797	4/1	4/1	98/49
	WARP	26568	-	-	82/324
	WELL	60158	2/8	-	26/2340
11	IDEAS	6512	19/1	8/1	18/3
	SHELLS	6434	3/3	-	4/320
	GRID	32025	37/1	4/1	172/1
	ROTOR	42060	2/3	-	44/321
	CAR1	95666	-	-	7/488
12	PIC3D	16442	9/1	4/1	180/90

Table 4: Geometry-related profile data

Type	Trace	Clear Cnt	Clear (000)	Prim (000)	Line Mean	Poly Mean
1	VOXEL	3	734	1074	33	3441
2	VASE0	1	1233	56	36	-
	CFD	1	1250	179	5	4006
	CAR0	2	2485	404	4	-
3	ROD0	1	968	68	20	-
4	STICK0	1	1244	92	21	-
5	COPTER	2	1929	121	-	588
6	BARC	1	1244	3518	-	1725
7	RAY	1	1261	510	65	757
	OILRES	1	1245	12145	312	8788
8	BALLS	2	2488	87	320	209
	LIGHTS	1	1243	472	-	579
	LATHE	2	1952	598	-	2695
	VASE1	2	2466	798	-	1056
	STICK1	1	1244	519	-	246
	BARS	1	1244	310	702	107
	FLIGHT	1	1313	1828	62	2038
	VORO	1	1244	1787	-	262
9	ROD1	1	968	256	-	308
10	SLICE	2	1889	374	202	38
	WARP	2	1486	303	-	12
	WELL	4	2460	137	123	2
11	IDEAS	2	2585	1156	3	220
	SHELLS	2	2552	1999	487	622
	GRID	2	2539	1159	3	58
	ROTOR	2	1964	517	511	13
	CAR1	2	2485	538	-	8
12	PIC3D	1	945	1328	35	87

Table 6: Pixel-related profile data

(CFD, RAY, and OILRES), runs were short due to per-primitive color or normal changes. The anomaly is LATHE, which contains both per-primitive and per-vertex normals.

Only six of the traces mixed primitive types and shading modes to a significant degree. Table 5 shows the fraction of transformed vertices due to each primitive type in these traces.

Type	Trace	Line	Poly	Tri
7	RAY	0.12	0.88	0.00
	OILRES	0.40	0.60	0.00
8	LIGHTS	0.00	0.80	0.20
	FLIGHT	0.07	0.92	0.00
11	IDEAS	0.07	0.00	0.93
	GRID	0.38	0.00	0.62

Table 5: Fraction of vertices for mixed primitives

Table 6 presents the pixel-related data for the example traces, which consists of the following items:

- The number of clear-viewport commands.
- The number of pixels written during viewport clearing.
- The number of pixels written during primitive rendering.
- The average number of pixels per line segment.
- The average number of pixels per polygon, including triangle strips.

Except in the case of the multi-window VOXEL trace, multiple viewport clears were due to separate clearing of the color and Z buffers. For the shaded scenes, the ratio of rendered pixels to cleared pixels can be used as a rough estimate of depth complexity[5]. This ratio is notably large for the BARC and OILRES traces, in which most of the pixels generated by the rasterizer are rejected by the Z-buffer or overwritten. These two applications would benefit significantly from better database culling.

6 Trace-based Benchmarking

Our benchmarking program, TBENCH, is simply a driver program for a GR interpreter. It begins by reading a complete trace into memory. It then runs a compiler over the trace to convert the vertex formats to the appropriate ones for the target machine. Next, TBENCH executes the trace's preamble to bring the GR interpreter to the appropriate initial state. Finally, TBENCH starts the clock, repeatedly executes the trace's frame for a minimum of ten seconds or ten updates, waits for the graphics system to be idle, and stops the clock. To insure fair comparisons, TBENCH generates a warning if the GR interpreter reports that the trace has used an unimplemented feature.

To date, we have implemented two GR interpreters. The GR-to-GL interpreter runs on all Silicon Graphics systems, and the GR-to-Starbase interpreter runs on the Hewlett-Packard TurboSRX. We have just completed the Starbase version, and while it is fully

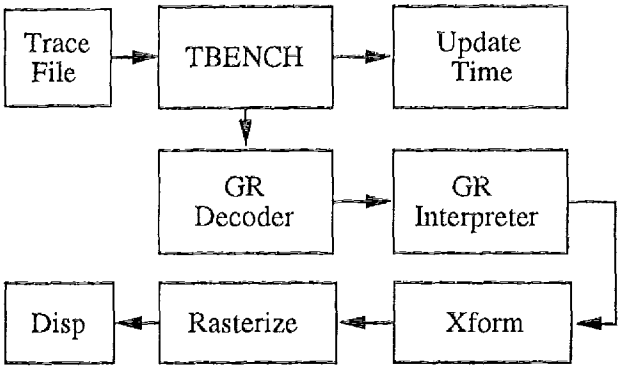


Figure 5: Trace-based benchmarking

functional, we have not had enough experience with it to insure that it is taking full advantage of the TurboSRX's performance potential. We will therefore present only the results from the GL version. Table 7 lists the specifications of the Silicon Graphics systems we tested.

Code	Model	Line/sec	Poly/sec
G	4D/70G	141000	4000
P	4D/25G	90000	5000
PT	4D/25TG	200000	20000
GT	4D/80GT	400000	55000
GTX	4D/220GTX	400000	100000

Table 7: Systems tested

Table 8 presents the update time achieved by each system on the 28 traces. A useful property of our benchmarking approach is that we can trace other benchmark programs. The COPTER trace is from the GPC picture-level benchmark (PLB) program[9]. We traced several PLB files and ran both PLB and TBENCH on all of the Silicon Graphics systems, obtaining identical results. We plan to use this technique on the TurboSRX to verify the efficiency of the GR-to-Starbase interpreter.

Table 9 presents the update-time performance of each system relative to the least expensive system, which is the 4D/25G. Note that the performance advantage of the high-end systems over the low-end systems varies greatly among the traces. This underscores the importance of application-specific performance evaluation.

Table 10 presents the relative performance of the systems as specified by the manufacturer. The differences in rendering speed between the systems tend to be diluted in real applications by the presence of viewport clears, buffer swaps, and window context switches.

Type	Trace	G	P	PT	GT	GTX
1	VOXEL	5956	5097	3177	1213	788
2	VASE0	52	50	50	34	34
	CFD	427	309	286	182	102
	CAR0	1744	2453	1333	769	417
3	ROD0	153	100	100	50	33
4	STICK0	648	385	254	118	118
5	COPTER	133	117	94	33	34
6	BARC	2300	1167	939	182	167
7	RAY	448	412	267	67	83
	OILRES	6775	4689	3895	714	723
8	BALLS	153	177	70	34	34
	LIGHTS	303	270	151	51	51
	LATHE	719	250	200	51	62
	VASE1	618	467	351	68	68
	STICK1	865	1136	384	116	116
	BARS	937	839	217	137	116
	FLIGHT	793	555	360	215	178
	VORO	2963	2272	1067	366	327
9	ROD1	566	301	266	51	34
10	SLICE	1525	1117	301	133	132
	WARP	3788	2002	517	250	250
	WELL	7325	3622	1034	467	476
11	IDEAS	1431	700	401	149	152
	SHELLS	1750	973	646	149	149
	GRID	5325	3269	1653	333	301
	ROTOR	7569	4434	1684	400	399
	CAR1	10588	6480	1836	750	603
12	PIC3D	10713	2869	-	263	250

Table 8: Update times in milliseconds

7 Other Uses

Our GR traces are a very convenient, compact, resolution-independent way to package a single frame of 3D graphics, and they have proven useful in areas other than performance analysis:

- User-interface experimentation: We have implemented a universal viewing program called TVIEW that reads a trace in binary or ASCII form and allows the user to "fly" through the scene. TVIEW provides a convenient, consistent way to test a system's "feel" using real data. When experimenting with new general-purpose user-interface devices and techniques, it is much easier to generate traces and work with TVIEW than it is to modify the original application programs. TVIEW supports a mouse-operated virtual trackball, the SpaceBall six-axis input device, and stereo viewing.
- Debugging: TVIEW's ability to directly read ASCII trace files makes it a convenient debugging tool. It is often helpful to capture a trace from a buggy program and use a text editor to modify the trace until TVIEW produces correct results. TVIEW has a reread-file feature that reduces turnaround time to one or two seconds, which is usually much faster than changing, recompiling, and relinking the original program.
- High-quality rendering: Using an iterative tiling technique, TVIEW can use a system's graphics hardware to render a trace to an image file at an arbitrary multiple of screen resolution. The combination of TGEN and TVIEW provides an easy way to generate publication-quality images from an arbitrary interactive 3D graphics program. We used TVIEW to generate the images in Plate 1.
- Testing: A library of traces can be used as a repeatable, realistic workload when doing software regression testing or production-line testing. We have implemented a program that executes a trace, computes a signature function on the resulting image, and compares this signature with a known-correct value.

Type	Trace	G	P	PT	GT	GTX
1	VOXEL	0.86	1.00	1.60	4.20	6.47
2	VASE0	0.96	1.00	1.00	1.47	1.47
	CFD	0.62	1.00	1.08	1.70	3.03
	CAR0	1.41	1.00	1.84	3.19	5.88
3	ROD0	0.65	1.00	1.00	2.00	3.33
4	STICK0	0.59	1.00	1.52	3.26	3.26
5	COPTER	0.88	1.00	1.24	3.55	3.44
6	BARC	0.51	1.00	1.24	6.41	6.99
7	RAY	0.92	1.00	1.54	6.15	4.96
	OILRES	0.69	1.00	1.20	6.57	6.49
8	BALLS	1.16	1.00	2.53	5.21	5.21
	LIGHTS	0.89	1.00	1.79	5.29	5.29
	LATHE	0.35	1.00	1.25	4.90	4.03
	VASE1	0.76	1.00	1.33	6.87	6.87
	STICK1	1.31	1.00	2.96	9.79	9.79
	BARS	0.90	1.00	3.87	6.12	7.23
	FLIGHT	0.70	1.00	1.54	2.58	3.12
	VORO	0.77	1.00	2.13	6.21	6.95
9	ROD1	0.53	1.00	1.13	5.90	8.85
10	SLICE	0.73	1.00	3.71	8.40	8.46
	WARP	0.53	1.00	3.87	8.01	8.01
	WELL	0.49	1.00	3.50	7.76	7.61
11	IDEAS	0.49	1.00	1.75	4.70	4.61
	SHELLS	0.56	1.00	1.51	6.53	6.53
	GRID	0.61	1.00	1.98	9.82	10.86
	ROTOR	0.59	1.00	2.63	11.09	11.11
	CAR1	0.61	1.00	3.53	8.64	10.75
12	PIC3D	0.27	1.00	-	10.91	11.48

Table 9: Relative performance

Prim	G	P	PT	GT	GTX
Line	1.57	1.00	2.22	4.44	4.44
Poly	0.80	1.00	4.00	11.00	20.00

Table 10: Specified relative performance

- Simulation: We implemented a program that uses a trace to drive a simulator for a new piece of graphics hardware. This enabled microcode developers to verify the correct execution of real application programs before actual hardware was available.

8 Future Work

Now that we have a reasonable way to measure *how* a system performs, we are concerned with how to find out *why* it performs the way it does. We are currently building a trace-driven graphics-performance simulator called TSIM. TSIM is a GR interpreter that simulates the performance of three stages of a graphics pipeline: traversal, transformation, and rasterization. The output of TSIM is an update-time prediction and a utilization factor for each pipeline stage (an indication of where the application-specific bottleneck is).

TSIM is intended to be general enough to simulate accurately the performance of a reasonably wide range of real graphics systems. To this end, it takes as a parameter a "system profile", which is a vector of numbers that tells each simulated pipeline stage how fast to be. We are implementing another program called SPGEN that generates a system profile for a real system. SPGEN is a portable, application-independent benchmark program that measures the performance of a system's graphics-pipeline stages by executing and timing small sequences of GR calls.

We plan to use these programs to conduct the following experiment:

1. Use TGEN to generate traces of real application programs.
2. Use SPGEN to generate profiles of real systems.
3. Plug the traces and system profiles into TSIM and generate stage utilization factors and an update-time prediction.
4. Test the accuracy of the update-time prediction by measuring the actual update time with TBENCH.

If TSIM generates accurate results, it will be a useful tool for understanding how graphics systems behave under real workloads. Our goal is to be able to answer questions such as, "If I make transformations twenty percent faster, which applications will see a reduction in update time?"

9 Summary

We have developed a convenient method for measuring how general-purpose interactive 3D graphics systems perform under real workloads. The method involves generating a trace file of rendering commands during the execution of an application program. The trace generator is convenient to use because it does not require any changes to user or system software and it does not affect the behavior of the program being traced. Traces are compact because they contain only a single graphics frame.

We have developed several tools that make use of the traces, including a dynamic profiler, an update-time measurement program, and a viewer. All of the tools are portable because they are built on top of a simplified immediate-mode graphics interface called GR. It is relatively easy to build an efficient GR interpreter on top of existing graphics libraries.

10 Acknowledgements

We would like to thank Jim Winget, Forest Baskett, and Paul Haeberli of Silicon Graphics for their good advice and generous support. We would also like to thank Susan Spach and Fred Kitson of Hewlett-Packard for helping us with the TurboSRX port.

References

[1] Kurt Akeley. The Silicon Graphics 4D-240GTX Superworkstation. *IEEE Computer Graphics and Applications*, 9(4):71–84, July 1989.

[2] Brian Apgar, B. Bersack, and Abraham Mammen. A Display System for the Stellar Graphics Supercomputer Model GS1000. *Computer Graphics*, 22(4):255–262, August 1988.

[3] Hewlett-Packard Company. *Starbase Reference*, 1988.

[4] Ardent Computer Corporation. Dore technical overview. April 1988.

[5] Nader Gharachorloo, Satish Gupta, Robert F. Sproull, and Ivan E. Sutherland. A characterization of ten rasterization techniques. *Computer Graphics*, 23(3):233–368, July 1989.

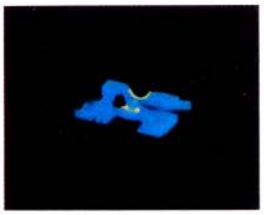
[6] Randi J. Rost, Jeffrey D. Friedberg, and Peter L. Nishimoto. PEX: A Network-Transparent 3D Graphics System. *IEEE Computer Graphics and Applications*, 9(4):71–84, July 1989.

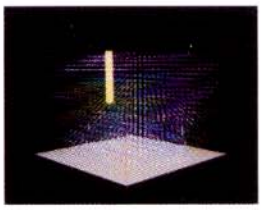
[7] Silicon Graphics, Inc. *GL Reference Manual*, 1988.

[8] Tektronix. *OnRamp Function List*, June 1989.

[9] S. Tice, M. Fusco, and P. Straley. The Picture-Level Benchmark. *Computer Graphics World*, July 1988.

Color images for this paper can be found in the color plate section.

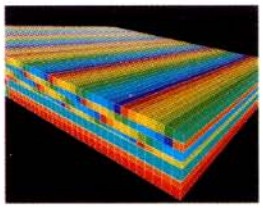

VOXEL

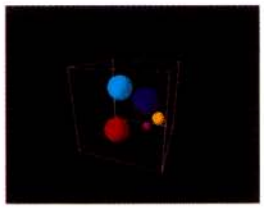

CFD


COPTER

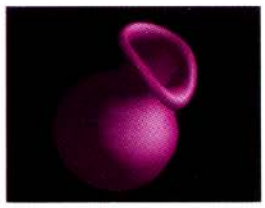

BARC


RAY


OILRES


BALLS

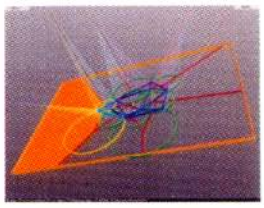

LATHE

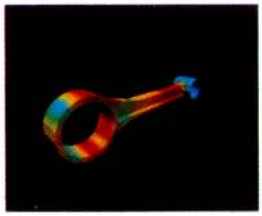

VASE1

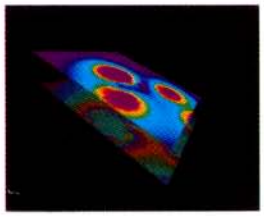

STICK1

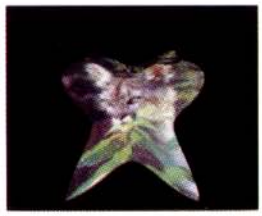

BARS


FLIGHT

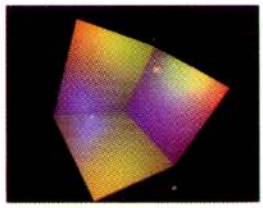

VORO


ROD1

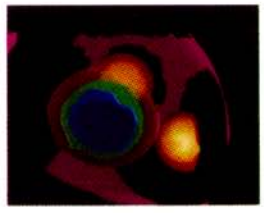

SLICE

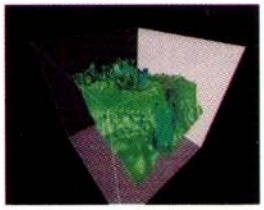

WARP


WELL


LIGHTS

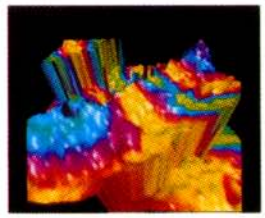

IDEAS


SHELLS


GRID


ROTOR


CAR1


PIC3D