A Simulation of A Graphics Subsystem

Aseem Agarwala & Antoine McNamara
University of Washington
548 Final Project Report

March 14, 2002

1 Introduction

We present a simple, complete, and extensible simulation of a hardware graphics subsystem. The architecture community relies heavily on simulation. Tools like Simplescalar [3] can be modified to help researchers predict the expected performance of new CPU designs, without the tremendous cost of implementing them in hardware. Up ntil now, these tools have simulated numerous CPU architectures, but have not taken into account GPUs (graphics processing units). Graphics acceleration cards are becoming standard on desktop hardware, and the applications that they aid are becoming increasingly essential. Even 3D games like Quake are considered important test benchmarks for new systems, and in fact it has been ported to simplescalar.

Our project is to create a hardware simulation of a combined CPU / GPU system, whose results could ideally be used to more accurately benchmark expected performance on real systems with graphics acceleration. With such a system, researchers could tweak various aspects of the pipeline and observe how it affects performance.

However, it is clear that is an ambitious task. The main problem is that there is a bewildering variety of options and choices to be made in the design of such a system. To understand this, it is first necessary to provide an overview of current and previous graphics subsystems.

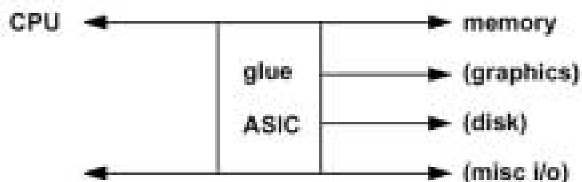

Figure 1: A simplified model of a computer.

2 Graphics subsystems

A simple model of desktop computers is shown in Figure 1 (Figures 1-4 are taken from [4]). Essentially, a computer conducts a large amount of communication between various components; common components of a computer include the CPU, memory, disk, network I/O, and others. These must all communicate requests and results to each other. While the CPU normally has a direct connection to memory, the other components reside on a shared bus. The most common bus today is PCI (personal computer interface), whose maximum bandwidth is 133 MB/s.

Thus, it is necessary to decide how a graphics subsystem fits into this picture. The trade-off is clear; fast graphics performance requires the transfer of large amounts of information between the CPU/memory and the GPU, but accomplishing this requires specialized designs that add cost.

2.1 GPU/CPU/memory organization

We show three common approaches to the problem.

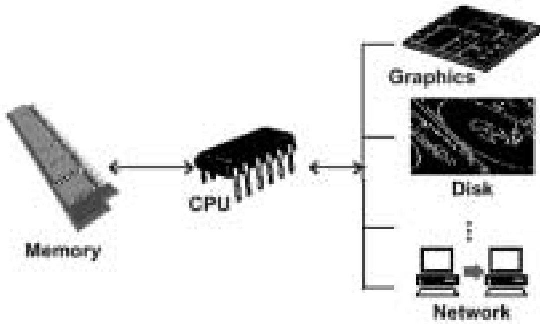

Figure 2: PCI Graphics Card.

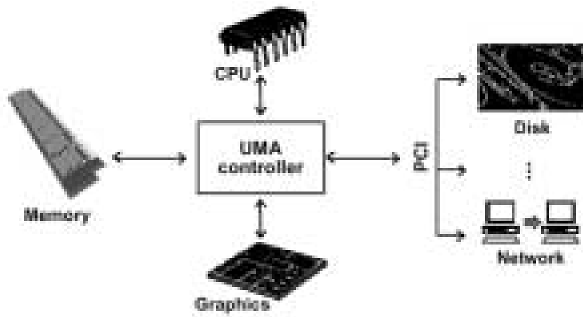

Figure 3: Unified memory architecture (UMA).

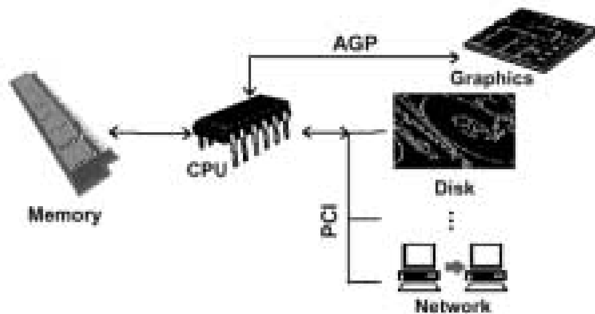

Figure 4: Advanced graphics port (AGP).

2.1.1 PCI graphics card

The cheapest, and until recently most common solution is to add a graphic card onto the PCI shared bus, as shown in Figure 2. This is a simple design that fits into existing desktop computers without any modifications. However, it has performance drawbacks. Any communication with the GPU must go over a shared bus, and thus must compete over a shared resource with other components such as the network which also communicate over PCI. Also, even without this competition, the bandwidth of PCI is not enough for demanding graphics applications.

2.1.2 Unified memory architecture

The unified memory architecture, or UMA, is an organization specifically for graphics workstations designed by SGI. As shown in Figure 3, all components communicate through a central controller. The individual communication links run at high bandwidth (500-800 MB/s), and the main memory (at 2.1 GB/s) serves as the memory for all components. In this way, there is less need for different components to pass around information; if the CPU wishes to pass a large chunk of memory to the GPU, it simply passes a pointer.

2.1.3 Advanced graphics port

The advanced graphics port, or AGP, is a dedicated bus between the CPU/memory and the GPU, as shown in Figure 4. It is designed by Intel for desktop computers. The bandwidth of this bus is 512 MB/s or 1024 MB/s, depending on whether information is passed on one or both edges of a clock cycle; either way, the bandwidth is significantly higher than PCI. Additionally, With an AGP communication to the GPU does not need to compete for a shared resource.

2.2 Other choices

We continue this discussion by detailing several other aspects of graphics subsystems and the choices they offer.

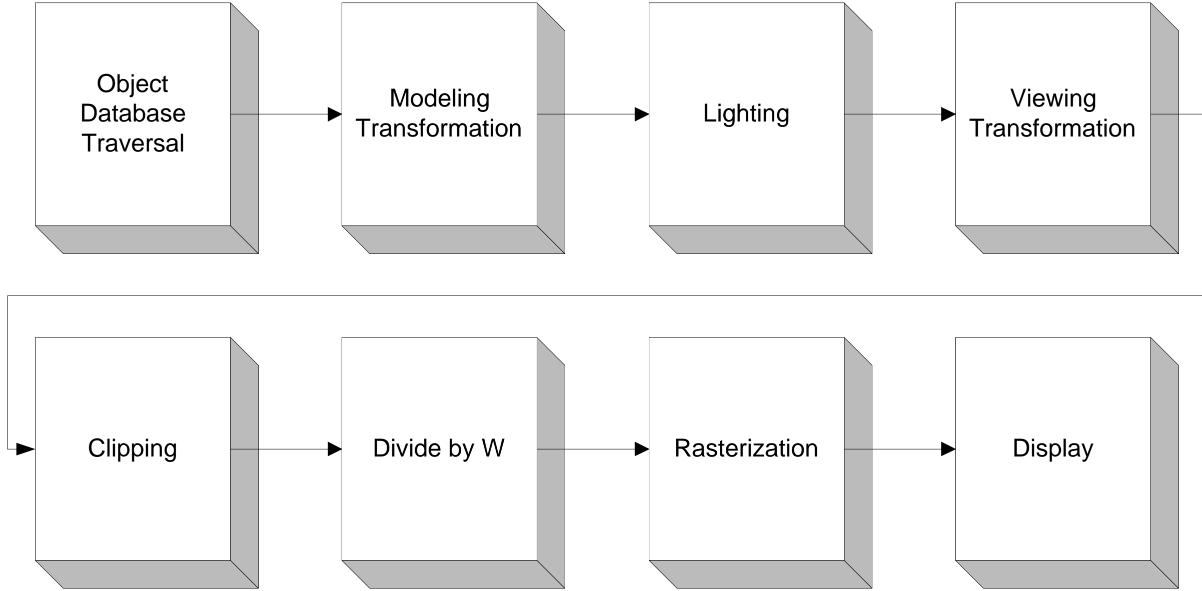

Figure 5: The rendering pipeline.

2.2.1 Graphics pipeline

The process of specifying and rendering three-dimensional models for computer graphics consists of several stages, as shown in Figure 5. The issue is that various graphics accelerator cards implement various amounts of this pipeline. Cheaper cards only perform parts of rasterization and display. Better ones include hardware-accelerated viewing transformations, clipping, and onwards in the pipeline. Recent cards like the market-leader NVIDIA GeForce3 also perform modeling transforms and lighting. SGI graphics workstations perform these tasks as well.

2.2.2 Card simulation

It would be most helpful to the computer architecture community to simulate a commonly use graphics card, such as the NVIDIA GeForce3. However, their technology is proprietary and they have not published enough details to write an accurate simulation. A recent publication by NVIDIA [8] describes a specific feature of the GeForce3; the user-programmable vertex engine. This allows developers to specify short sequences of custom assembly instructions that will be performed per vertex. However, this only describes a specific, new feature for modeling transformation and lighting, which is only a fraction of the rendering pipeline. Thus, it was not enough to simulate the GeForce3.

SGI, on the other hand, has published several papers on their architectures [1, 2, 7], making it easier to simulate their hardware approaches.

2.2.3 Asynchronous execution

A further complication is presented by the fact that CPU's and GPU's do not typically operate on the same clock; they run asynchronously. Because of this their communication must be buffered, which is done through the use of a first-in / first-out (FIFO) queue. If the CPU hands data at a quicker rate than the GPU can handle, the FIFO fills up and the GPU sends an interrupt to the CPU; in this case, the application is considered GPU-bound. Alternatively, if the GPU exhausts the FIFO and stalls waiting for more data from the CPU, the application is considered CPU-bound.

This is clearly difficult to model within existing architectural simulation tools. One could take an existing simulator like Simplescalar and make it multi-threaded, so that the CPU and

GPU run through separate threads in parallel. However, beyond the obvious difficulty of this, there is no reason to believe that the way an operating system schedules threads would accurately model the asynchronous execution of the CPU and GPU.

2.2.4 Minutiae

Several other choices are worth mentioning. For one, it is most useful to model commonly used graphics application programming interfaces (API's). The two most common are OpenGL and DirectX.

Finally, a relatively new feature of graphics accelerator cards are programmable engines, such as vertex shaders and pixel shaders. These allow developers to specify short sequences of custom assembly instructions that will be performed per 3D vertex or per fragment in the rasterizer. These are powerful new features, but are complex systems to simulate.

3 Design choices and simplifications

Now that we have presented the large set of options in simulating graphics subsystems, it is necessary to make choices. Clearly, modeling all aspects of such a system to its full complexity is a prohibitively large task; we thus are forced to make simplifying assumptions.

- We chose to model a small portion of the OpenGL API; OpenGL is a very large API [9], and to model it in its entirety would be too large a project by itself. We thus implemented enough of the API to render 3D triangle meshes and 3D lines. Also, we chose to ignore programmable engines; however, a goal of our design was to leave the system open to this addition at a later time.

- Since our target is desktop computers, we choose as a hardware model the published SGI architecture most relevant to this goal [7]. This architecture was shipped in the IRIS Indigo Extreme.

- We separate the graphics pipeline between the CPU and GPU as follows: the CPU performs modeling transformations and lighting, while the GPU calculates the rest of the pipeline.

- As far as memory organization, we simply assume instantaneous transfer of data between the CPU and GPU. This is not significantly different from the effect of UMA or AGP; however, it does not model the latency of the bus or the possibility of exceeding its bandwidth.

- As far as asynchronous execution, we model the timing of the CPU and GPU separately. Thus, the simulation of the CPU suspends during the simulation of the GPU, and the simulation of the GPU suspends during simulation of the CPU. This is equivalent to assuming that the components run in parallel and that one component never forces the other to stall. That is, the CPU timing data assumes the application is CPU-bound, and that the GPU never interrupts the CPU. The GPU timing data assumes the application is GPU-bound, and that it never has to wait for the CPU to provide data.

4 System Overview

A dataflow overview of the system we built can be seen in Figure 6. Our system takes as input a **nff** file (Neutral File Format) which describes the three-dimensional geometry of a model in the form of triangles.

4.1 Lighting

This file is opened by our *user application*. Since modeling and lighting are not implemented in hardware, it is the task of the user application to light the model. Details of lighting can be found in [6]; we provide a quick overview to give a sense of the volume of floating point calculations involved.

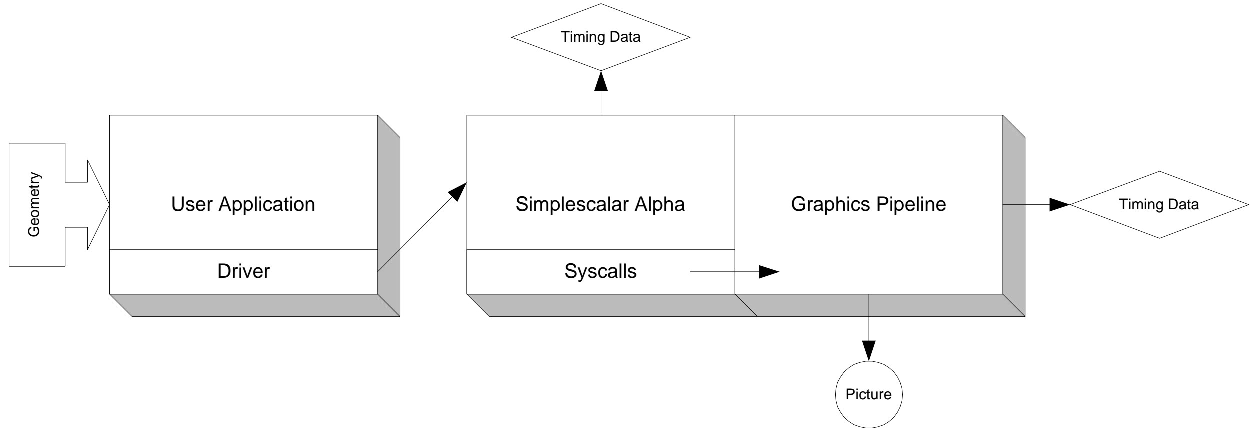

Figure 6: An overview of our system.

The application must first compute normals at each vertex (a normal is a vector perpendicular to the model surface). Normals are computed by iterating over each triangle in the model, computing the normal to the triangle (calculated as the cross product of two sides of the triangle), and adding this normal to each vertex normal. Each vertice normal is then normalized (divided by the magnitude of the vector, which involves a square root).

Once normals are calculated, we use an ambient and diffuse lighting model to illuminate 3D models; this illumination simply consists of calculating a color for each vertex. Ambient light adds a constant to this color. Diffuse light adds the dot product of the light direction and the normal of the model surface.

Once lighting is calculated, a data structure consisting of a list of triangle is created. Each triangle has three vertices, and each vertex has six floats (three for location, three for color). This data is then passed to the GPU using OpenGL calls.

4.2 Passing data to the GPU

Once graphics data is created, it must be passed to the GPU. The interface to the user consists of OpenGL calls; we implement eight individual calls in our *driver*. The driver is compiled directly into the user application, and both are compiled into Alpha instructions using a GCC cross-compiler.

Our CPU/GPU simulation is built directly onto sim-alpha [5], a version of Simplescalar which accurately models the DEC Alpha. So, a mechanism must be created to pass graphics data from the user application through Simplescalar directly to the graphics pipeline.

This mechanism is the *syscall*. Syscalls allow the CPU simulation to trap system calls and execute them; common syscalls handle file I/O and standard I/O. We thus add seven additional syscalls to Simplescalar, which interact with the graphics hardware simulation. The graphics pipeline is then compiled directly into Simplescalar.

We now trace the path of graphics data from the user application to the GPU. First, the user application executes OpenGL calls. The driver implements these calls using the *asm* command, which is a GCC extension to C that allows direct specification of assembly commands. The appropriate opcode for the syscall we wish to execute is placed in register $A0. Then, any arguments (up to four) are placed in registers $A16-$A19. Finally, the Alpha assembly instruction *callsys* is executed.

We pass at most one argument, which is a pointer to floating point data. Within the syscall handler in Simplescalar, we add handlers for the graphics syscalls. If data is being passed, the pointer to floats is used to copy the data from the simulated user application to the GPU. In

this way, the user application can communicate directly to the GPU.

4.3 OpenGL calls

Along with the basic OpenGL calls used to pass triangle, vertex, and line data to the GPU, several matrices for viewing transformation and projection are calculated within the driver. One such call is **gluPerspective()**, which allows the programmer to specify the details of the projection of the model from 3D to 2D. Another command is **gluLookAt()**, which is an intuitive means for the programmer to specify the viewing transformation so that the camera is located correctly and pointing at the model. These calculated matrices are passed to the GPU.

5 The graphics pipeline

The last, and largest, step is to model the rendering pipeline. Overall, we used the SGI chip as a foundation, but there are several aspects that were still unknown, unfeasible, or simply dated. In these cases, we used our best judgement, with the intention of creating a general model for a graphics hardware pipeline that could be easily extended in the future to more specific modern architectures as needed.

The SGI graphics chip has an 8-way SIMD parallel front-end (viewing transformation and clipping) and a hyper-pipelined back-end (rasterizing). Because of the SIMD nature of the parallelism, the chip can get excellent parallelism (up to 8 times speedup) on groups of primitives of the same type. On the other hand, because it cannot calculate primitives out of order, it loses all parallelism in the case of alternating triangles and lines! This is why we opted to support `glBegin(GL_LINES)` as well even though we were much more concerned with triangle performance: not supporting line primitives would always result in an unrealistic 8-fold speedup.

The chip is broadly divided as shown in Figure 7. When the chip gets a call to begin processing, the FIFO contains triangle and line primitives from the CPU. The Command Parser removes the primitives in order from the FIFO and distributes them to the 8 Geometry Engines (GEs). Here, all processors run in lock step, projecting and clipping the primitives. Once they complete, the primitives sit in the GE output buffers and are fed one by one into the rasterizer pipelines (one pipeline draws the even lines, the other the odd lines) and they are drawn to the framebuffer. Our simulation has the same structure, although it runs sequentially. It loops until the FIFO is empty, each time distributing the oldest elements in the FIFO to GEs, executing the GEs, and calling the Rasterizer on the GE output buffers.

5.1 The Command Parser

The GPU FIFO is implemented as a linked-list. In reality, the graphics card sends signals back to the CPU when the FIFO fills up, but in our simplification the FIFO can be as large as possible and will be emptied completely on a call to `GPU_Draw()`. Because we wanted the number of GEs to be variable, the GE input buffers are also represented as a linked-list, with one node per SIMD processor. The command parser moves primitives from the FIFO to this list until it sees a different type of primitive or has seen `NUM_GE` elements. The module returns the number of elements that were distributed for timing analysis and to determine when the FIFO is empty.

5.2 Geometry Engines

The GEs are responsible for projecting and clipping primitives. In hardware, they are just simple processors with a multiplier and an adder and run different series of microinstructions depending on the type of primitive (line or triangle). We don't have any information on these micro-programs, but assume they use the traditional algorithms presented in [6] optimized for the particular hardware. Our software pipeline also uses these same algorithms.

First, the viewing matrix is applied to all vertices in the primitive (simple matrix multiplies). Second, the primitive is clipped to the `zNear` and `zFar` z-planes specified in `gluLookAt()`. Third,

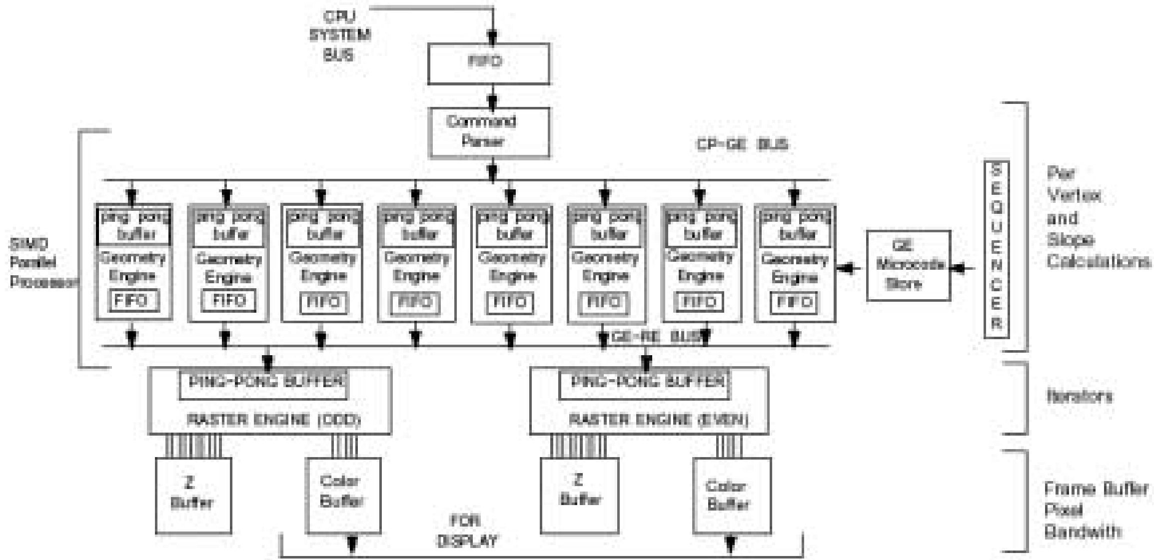

Figure 7: An overview of the modeled SGI graphics chip.

the projection matrix is applied to all vertices, mapping to screen coordinates. Last, primitives are clipped in x and y to the screen extents. The clipping against arbitrary planes in 3D is expensive, and so the x and y clipping are pushed back after the projection, using Cohen-Sutherland viewport clipping [6].

Clipping a triangle to a plane can result in either a triangle or a quadrilateral, but since we can only rasterize triangles, this latter case must be split. Since each primitive must be clipped against six planes, the number of primitives handled by a single GE can grow tremendously. We keep each primitive as a linked list, initally with one node. If the primitive is clipped completely, the node is removed. If it must be split, a new node is created. All the operations (projection, clipping, rasterization) are implemented to work on arbitrary sized lists. In hardware, when one GE needs to do more work than another, the others stall, so we consider this when doing timing evaluation.

5.3 Rasterizer

While the GEs are small processors that run micro-instructions, the rasterizer is dedicated hardware, pipelined into 26 stages between GE output and the framebuffer. Our simulator scan-converts lines using the Midpoint Line Algorithm [6]. Triangles are rasterized using the technique described in the SGI paper:

1. Calculate the slopes of the three sides, in terms of change in x over a unit change in y,
2. step through the major edge of the triangle (the edge between the maximum and minimum y points) and the two minor edges to generate scanlines through the triangle,
3. and draw each scanline, linearly interpolating the colors (Gouraud shading) and depth.

For each pixel drawn to the screen, compare the depth to the corresponding value in the z-buffer, and, if closer, overwrite the frame buffer and z-buffer.

Rasterization algorithms get tricky in the case of subpixel primitives, especially triangles.

When the major edge of the triangle is entirely within a pixel, the slope calculation may lead to inappropriately large scanlines and other numerical issues may crop up, leading to numerous special cases to catch. We handle most of these, but some still create artifacts (as may be evidenced along the silhouette of the teapot). We don't feel these would be too difficult to fix, but other aspects were more pressing and we were still pleased with the results.

6 Timing

The goal of our project was to provide a simulation that might easily be generalized to a number of graphics cards; therefore, we did not focus on producing accurate timing data for the particular chip we implemented. Even if we wanted to, our reference papers weren't specific enough to provide such insights, especially for the variable length Geometry Engine calculations. Instead, we've laid the foundation for a timing system that could trivially be extended to produce accurate results for our SGI chip given the appropriate constants, and but that is still general enough to be applied to another card with similar structure.

In hardware, the three main subsystems are pipelined: the command parser, the GEs and the rasterizer all run simultaneously. The Geometry Engines are always the bottleneck in the pipeline, and so the total rendering time can be modeled as the time it takes the command parser to distribute the *first* batch of primitives, plus the total time for GEs to execute on *all* the primitives, plus the time to rasterize the *last* batch:

$$TIME_{ALL} = CP_{FIRST} + D_{ALL} + R_{LAST} \quad (1)$$

The command parser takes time linear in the number of primitives it distributes, and we define a constant `NUMCYCLES_TO_DISTRIBUTE` to allow the user to change the specifics. On the other end, the number of cycles the it takes to rasterize a group of primitives is the pipeline depth (26 for our chip) plus a single cycle for each extra primitive.

Model	Immediate	Display Lists
two-triangles	756	521
teapot	221309	14126

Figure 11: Graphics pipeline rendering time in cycles.

The timing on the GEs is more complicated, since they run in lockstep. When one GE needs to do extra work, the others stall, but we cannot merely take the maximum time required for any one particular GE, since they *all* might have stalled at one point or another. Instead, we simplify the problem at bit, and divide the variable work into 7 segments: the 6 clippings and the projection transformation (the viewing transformation is always only applied to a single primitive and hence is constant time for all GEs). For each GE execution, we record the amount of clock cycles required for that particular segment and then take the maximum for each segment across all GEs, finally summing these maximums to obtain the total running time for the block. This still might miss some subtle interactions between SIMD processors within a single segment (e.g., while clipping in x, one GE stalls on the first triangle, another on the second), but modeling these interactions would make the timing analysis much more complex and it's not obvious how influential these extra interactions would be. Because the number of cycles required for each GE segment is dependent on the micro-code programs they execute, we have defined the parameters as constants and leave them to be filled in by someone with more intimate knowledge of the chip's structure.

7 Results

We show two rendered results from our system, along with timing data. In Figure 8 we see a rendering of two triangles. While the triangle count is small, this example shows a case of complicated clipping; each triangle must be separated into five triangles before rendering. This example also shows Gouraud shading. Our second example is the classic Utah teapot, a common ob-

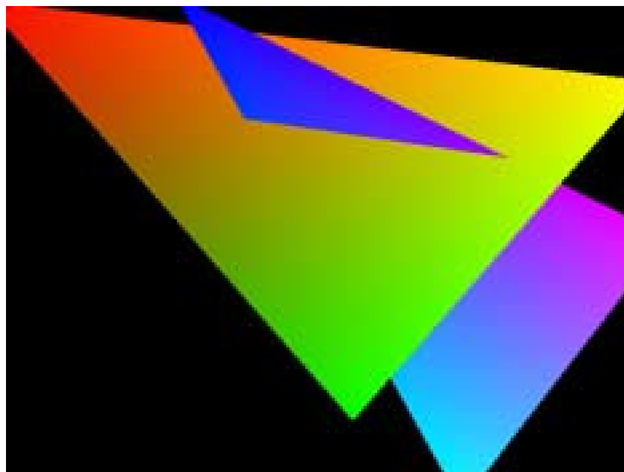

Figure 8: Two rendered triangles.

ject used to test computer graphics systems. It is shown in Figure 9, and consists of 3751 triangles. We use a simple gray color to best indicate shading. The user application required 52,754,990 cycles to specify and light the model (a partial listing of sim-alpha output is given in Figure 10). At 500 MHz, this would require about .1 seconds. Of course, much of this time came from computing normals, which must be done only once. Lighting must be calculated every time the object moves relative to the light source, and so our numbers indicate this could be done interactively (interaction generally requires 15 frames per second). Clearly though, moving lighting to hardware would improve speed for larger models.

For the graphics pipeline, we generate two types of timing data, corresponding to two common modes in OpenGL. One is *immediate-mode*, where primitives are rendered as they are specified; this is slower since the hardware cannot take advantage of parallelism. Another is *display lists*, where a large list of primitives is passed at once to the pipeline. This mode is faster, but the geometry cannot be modified as easily. The timing data, in cycles, is shown in Figure 11. Clearly, these models can be viewed at much faster than interactive rates.

References

[1] Kurt Akeley. Reality engine graphics. In *Proceedings of ACM SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 109–116, August 1993.

[2] Kurt Akeley and Tom Jermoluk. High-performance polygon rendering. In *Computer Graphics (Proceedings of ACM SIGGRAPH 88)*, pages 239–246, August 1988.

[3] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.

[4] Keith Cok. Developing efficient graphics software: the Yin and Yang of graphics. In *SIGGRAPH course program*, 2000.

[5] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 266–277. IEEE Computer Society and ACM SIGARCH, June 30–July 4, 2001.

[6] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, 2nd edition, 1990.

Figure 9: A rendered Utah teapot.

[7] Chandlee B. Harrell and Farhad Fouladi. Graphics rendering architecture for a high performance desktop workstation. In *Proceedings of ACM SIGGRAPH 93*, Computer Graphics Proceedings, Annual Conference Series, pages 93–100, August 1993.

[8] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 149–158. ACM Press / ACM SIGGRAPH, August 2001.

[9] Mason Woo, Jackie Neider, Tom Davis, and OpenGL Architecture Review Board. *OpenGL programming guide: the official guide to learning OpenGL, version 1.1*. Addison-Wesley, Reading, MA, USA, second edition, 1997.

```
sim: ** simulation statistics **
sim_num_insn              54165059 # total number of instructions committed
sim_num_refs              18807013 # total number of loads and stores committed
sim_num_loads             14005170 # total number of loads committed
sim_num_stores        4801843.0000 # total number of stores committed
sim_num_branches           8635973 # total number of branches committed
sim_elapsed_time               327 # total simulation time in seconds
sim_inst_rate          165642.3823 # simulation speed (in insts/sec)
sim_total_insn            59433502 # total number of instructions executed
sim_total_refs            20292912 # total number of loads and stores executed
sim_total_loads           15224678 # total number of loads executed
sim_total_stores      5068234.0000 # total number of stores executed
sim_total_branches         9103585 # total number of branches executed
sim_cycle                 52754990 # total simulation time in cycles
sim_IPC                     1.0267 # instructions per cycle
sim_CPI                     0.9740 # cycles per instruction
sim_exec_BW                 1.1266 # total instructions (mis-spec + committed) per cycle
sim_IPB                     6.2720 # instruction per branch
Onbus.idle                  0.9888 # fraction of time bus is idle
Onbus.queued                0.1886 # average queueing delay seen by bus request
Onbus.requests              232820 # number of transmissions on bus
Onbus.idle_cycles         52163725 # number of cycles bus was idle
Onbus.queued_cycles          43899 # total number of queued cycles for all requests
Membus.idle                 0.9974 # fraction of time bus is idle
Membus.queued               4.8792 # average queueing delay seen by bus request
Membus.requests              13810 # number of transmissions on bus
Membus.idle_cycles        52616904 # number of cycles bus was idle
Membus.queued_cycles         67382 # total number of queued cycles for all requests
SDRAM.accesses                6904 # total number of accesses
ld_text_base          0x0120000000 # program text (code) segment base
ld_text_size                212992 # program text (code) size in bytes
ld_data_base          0x0140000000 # program initialized data segment base
ld_data_size                 75936 # program init'ed `.data' and uninit'ed `.bss' size in bytes
ld_stack_base         0x011ff9b000 # program stack segment base (highest address in stack)
ld_stack_size                16384 # program initial stack size
ld_prog_entry         0x0120000300 # program entry point (initial PC)
ld_environ_base       0x011ff97000 # program environment base address address
ld_target_big_endian             0 # target executable endian-ness, non-zero if big endian
mem_brk_point           0x40014000 # data segment break point
mem_stack_min           0x1ff97000 # lowest address accessed in stack segment
mem_total_data                 75k # total bytes used in init/uninit data segment
mem_total_heap           -4194297k # total bytes used in program heap segment
mem_total_stack           4194321k # total bytes used in stack segment
mem_total_mem                  99k # total bytes used in data, heap, and stack segments
```

Figure 10: Partial timing results from sim-alpha for teapot rendering.