Gate-Level Implementation of a Hardware 3D Accelerator

6.837 – Introduction to Computer Graphics – December, 1999

Abstract:

This goal of this project was to design, implement, and simulate a hardware 3D accelerator. This accelerator performs basic setup, rasterizing, shading, texturing, blending, and Z-buffering of the pixels at a very high rate of up to four pixels every clock cycle. Clocked at a reasonable 150 MHz, the system outperforms many current high-level consumer graphics accelerators. The design involved two major phases: the first to design the hardware description language and simulator, and the second to design and implement the accelerator itself.

Designer:

Andreas Sundquist (asundqui@mit.edu)
Junior in courses 6-2 and 8.

Introduction:

Designing a hardware 3D renderer is a considerable task. The algorithms at hand, from a software point of view, are relatively simple to implement. With today's powerful machines, it is even possible to implement them so that they run in real-time with low-complexity inputs. However, a software solution will always be several orders of magnitude slower than a hardware solution. There is an increasing trend for computers these days to ship with 3D accelerators since people's demand for realistic 3D graphics is insatiable. This demand is also pushing the development of faster, more sophisticated, more complicated accelerator chips. For example, the latest entry in to the Wintel market, NVidia's GeForce256 graphics chip consists of 23 million transistors – compare that to Intel's latest processor which has about 28 million transistors. Graphics accelerators are fast approaching or even exceeding the size and complexity of the CPU, and in general the graphics accelerator has a far greater MFLOPS rating than the CPU.

A hardware implementation of standard raster algorithms is much more complicated than a software one for many reasons. For one thing, there isn't necessarily a standard "library" of functions or operators. These have to built from scratch, which requires a complete understanding of the functions down to the bit level. Second, operations are not performed serially as would be typical for a software-written algorithm. Though this acts in our benefit by allowing for massive parallelism, it also means that we need to understand the exact state of the entire system at any point of time, not just its state relative to the neighboring "instructions". Third, simulating and debugging is troublesome for hardware because it's much more difficult to understand the dynamics of how the entire system changes at the same time for every clock, unlike the familiar source-level tracers. Thus, although the base features I plan to include would be too simple if this were a software project, as a hardware implementation, the base features will prove very challenging.

One development in hardware design technology is the emphasis on integrated ASIC chips. Increasingly, what used to take several circuit boards with a myriad of different chips is now being integrated into single-chip solutions. This integration allows for a much larger number of gates, much higher speeds, and ultimately for much greater functionality. In the context of a single integrated chip, the problem domain for designing a 3D accelerator is mostly restricted to digital logic, and modeling such a system can be done very effectively in software. Since software designs are always cheaper and easier to built and prototype than hardware designs, a hardware designer will typically complete the entire system in software before actually producing any hardware.

Goals:

The ultimate goal of this project was to create a model describing a hardware system that, when simulated, renders three-dimensional images. I did not want to use a standard hardware description language like VHDL or Verilog because I needed to have greater control and efficiency in the simulation, and because I wanted a lot more flexibility in the semantics of the description. So, the design of this project actually involved two major components: the design of this hardware description language, and the implementation of the hardware accelerator in this language.

The first part of the project, although it took a considerable amount of extra work that was not graphics-related, was fairly critical to this project. I made the decision to create the “language” as a subset of C++ programming code, written in a way whose structure similar to other languages. That way, I would make use of the C++ parser and compiler, allowing for much faster execution of the simulation, and also much greater flexibility in the construction of the modules. For example, to create a barrel shifter, I would only have to implement one such generic module that generates the internal structure for any given input sizes algorithmically in C++, a very powerful feature. Also, once I created the floating point arithmetic modules and verified that it worked, which were fairly complex, a full bit-wise simulation of the modules would no longer be necessary – I would simply make use of the floating point operations available in the compiler and by the computer.

For the actual hardware 3D accelerator, I defined a basic feature set that I would implement, in addition to extra features that I wanted to add if I had time. The system should be able to setup and rasterize convex polygons, have a peak rendering speed of at least one pixel per clock (hence, it must be fully pipelined), perform basic shading (Gouraud interpolation), output to a 32-bit RGBA frame buffer, make use of a Z-buffer, do trilinear mipmapped texturing, and be able to composite pixels (alpha values). Some extra features I thought about including were some pixel parallelism, multitexturing, datapath/bandwith optimizations, caching, antialiasing, 2D acceleration, standard video output, and acceleration of the geometry, transform, and lighting stages. In the end, the extra features I decided to push for were pixel parallelism and bandwidth optimizations. My goal was to have the accelerator be able to continuously output four fully textured, shaded, and blended pixels every clock cycle. This would rival the performance of the top-of-the-line graphics accelerators in the market today.

My final goal was to produce a few sample images to illustrate the capabilities of my accelerator. Otherwise, I would have no “results” to show, other than massive amounts of code and a program that updates a cryptic textual display of the state of some of the modules.

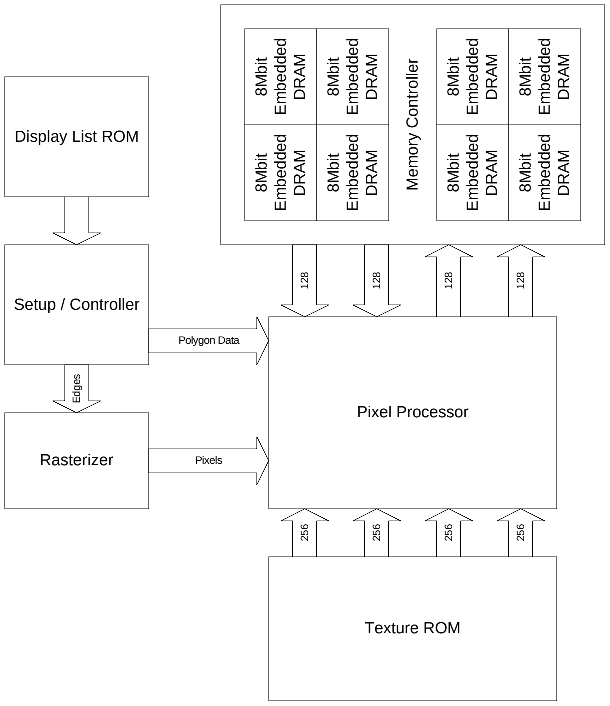
Achievements:

The most significant achievement of this project is, of course, that the hardware model does indeed produce 3D output images when simulated. I managed to satisfy the basic feature set I proposed, as well as incorporate some of extra performance features that makes the system at least as fast as accelerators sold on the market today. A more in-depth analysis of the achievements follows:

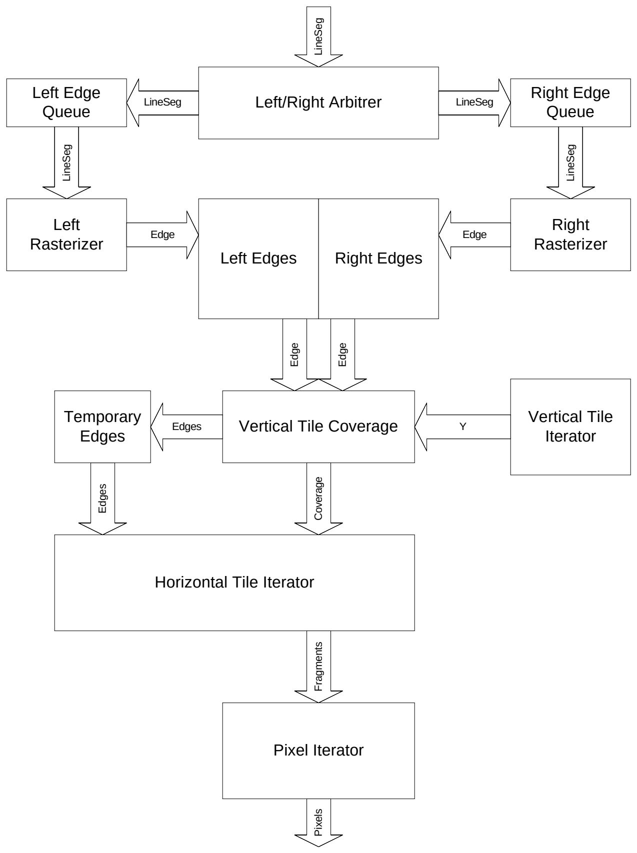
- Hardware simulator: This was no trivial task – indeed a very large portion of work done for this project went into designing an efficient and powerful hardware description language. Using two base objects, Module and Net and deriving other modules and nets from there, making use of C++’s object oriented programming methodology, I created a system that allows me to describe the hardware similar to the typical form of an HDL in addition to the extra power in algorithmically generating objects. An example of the language follows – this generates a generic barrel-shifter algorithmically:

```
Shr::Shr(int gateWidth, Net *A, Net *Count, Net *X) : SuperModule()
{
    assert((A->width>=gateWidth) && (X->width<=gateWidth));          (Enforce net input/output widths)

    InNet *inA = new InNet(A), *inC = new InNet(Count);              (Bind and register inputs)
    AddIns(2,inA,inC);

    OutNet *outX = new OutNet(X);                                    (Bind and register outputs)
    AddOuts(1,outX);

    Net *a = inA;                                                    (Keep track of intermediate nets)
    int i = inC->width-1;                                            (Shift-count bit counter)
    DWORD c = (DWORD)1<<i;                                           (Bit shift for current shift-count bit)

    for ( ; i>=0; --i, c >>= 1) {                                    (Iterate over each bit in the shift-count)
        Net *b = new Net(gateWidth);                                 (Create two new nets)
        Net *s = new Net(1);
        AddNets(2,b,s);
        Net *a2;
        if (i>0) {
            a2 = new Net(gateWidth);                                 (Not the last bit, create a temp net)
            AddNets(1,a2);
        } else
            a2 = outX;                                               (Last bit – output goes to X)
        Bitsel *bits = new Bitsel(b,a,(int)min(c,gateWidth),         (Create shifted input)
            (int)(gateWidth-min(c,gateWidth)),NetZero,0,(int)min(c,gateWidth),NULL);
        Bitsel *sel = new Bitsel(s,inC,i,1,NULL);                    (Isolate current shift-count bit)
        Mux *mux = new Mux(gateWidth,s,a2,2,a,b);                    (Mux between shifted and non-shift inputs)
        AddModules(3,bits,sel,mux);                                  (Register the sub-modules)
        a = a2;                                                      (Update intermediate net)
    }
}
```

- Display-list driven rendering architecture: The below diagram is an overall picture of the components of the system and how they interact. The setup controller reads data from the display list, updates the polygon information, and sends the line segments to the rasterizer. The rasterizer outputs pixels to the pixel processor, which renders those pixels using information from the setup controller and texture data to update the frame buffer. Since the display list and the texture memory will not need to be modified during the rendering of an image, they are ROMs, simplifying their implementation. In addition, the texture ROM is fairly idealized in that it can supply all the texture data needed by the pixel processor without ever stalling. As I'll explain later, this wouldn't be true in a real implementation, but I leave caching issues outside the scope of the project.



- Convex polygon rasterizer: This unit is double-buffered so that while the pixels of one polygon are being drawn, it can rasterize the next polygon, eliminating most of the delay between successive polygons. It takes in line segments and queues them as left or right edges depending on their orientation, and simultaneously operating left and right-edge rasterizers scan convert one row of a left edge and one row of a right edge every clock. Thus, the total number of clock cycles to rasterize a polygon is roughly equal to the number of scan lines that intersect the polygon. This way, even for skinny vertical polygons, the rasterizer will not take longer than the it takes to draw the actual pixels. An additional feature of the rasterizer is that it outputs pixels in tile-order, avoided the repeated cost of crossing page boundaries in the frame buffer. Below is a block diagram of the functioning of the rasterizer:

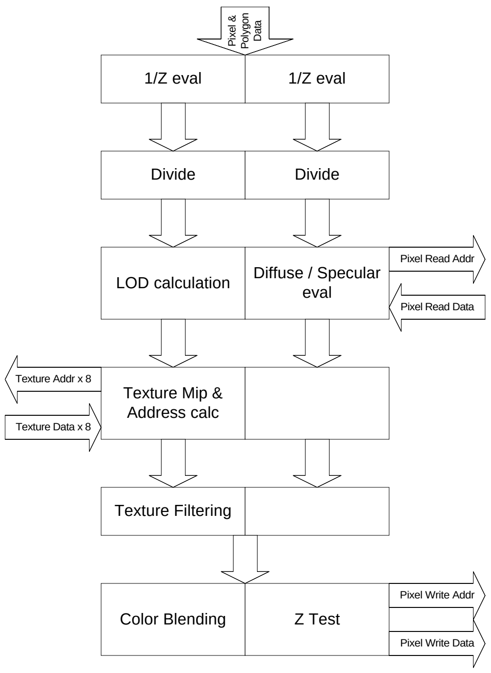


- Floating point data: Floating point numbers were used for simplicity in abstracting the data types, as well to avoid scale and precision problems. I designed an extensive library to work with floating point types, including addition, multiplication, division, and integer conversion. The performance impact of using floats instead of fixed point integers was not very significant. I designed each operation to work as a combinational circuit, running in time less than the clock period, including division, which I implemented using a small table and an iteration method.

- Shading: I made use of plane evaluators to accurately determine the shading parameters at each pixel of the polygon. In addition to a diffuse color component, I added a second "specular" color component. This way, the diffuse color could be blended with the texture, and then an extra "specular" term added to that. In addition, the shading parameters are interpolated in a perspective-correct fashion by dividing them by the (1/z) value for each pixel, which is also determined by an evaluator.

- Texturing: The architecture accepts 32-bit RGBA mipmapped textures and trilinearly filters the textures. In other words, it calculates the base-2 log level of detail at each pixel and selects the two adjacent mipmapped levels. Then, it bilinearly filters the four texels surrounding the texture coordinate for each of the levels, and interpolates between the two mipmap levels. The architecture assumes that it can read eight texels per pixel without stalling, which is clearly not always true. With a fully associative texture cache, and not-too-detailed textures, the texture-fetching stage probably would stall the pipeline less than 10% of the time, however.

- <u>Quad-pixel pipeline:</u> The peak rendering rate of the system is four pixels per clock. At 150Mhz, this translates into 600 million pixels drawn every second, which is a higher rating than for all the consumer-level PC graphics accelerators today. Of course, it doesn't always operate at full efficiency, for example because of frame buffer and texture memory read/write conflicts or stalls, and also because of the alignment of data in memory. I have computed that the alignment problem will, on average, degrade performance by around 10%. To keep the four pixel pipes running, there are some fairly massive datapaths in the system. For example, the interface between the frame buffer and the pixel processors is 512 bits wide. This was necessary since each pixel requires 64 bits of storage, and it needs to read 4 pixels and write 4 pixels every cycle. With simultaneous reads and writes, I had find a way to map pixel coordinates to memory addresses in such a way to minimize conflicts. Below is a diagram of the pixel rendering pipeline:

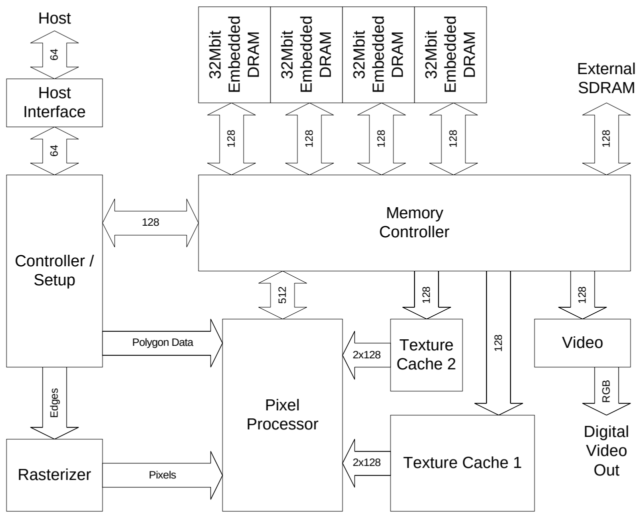


- <u>Floating point, 24-bit Z-buffer:</u> A floating point z-buffer is clearly preferable to an integer one because it allows the discrete depth values to be spread over a very large range, and reduces the resolution with distance, as desired.

There were many significant obstacles I had to overcome for this project:

- <u>Size of the project:</u> Yes – it was a *lot* of work. I severely underestimated the amount of work that would be necessary to implement the system. Figuring out the dynamics of the architecture was quite a challenge, since the various components are so interdependent. Jolt and sleep deprivation were the only ways to overcome this problem.

- Compiler limitations: I used Borland C++ 3.1, which is many years old now. Hence, I was restricted to the sub-640K region of memory. This presented a difficult constraint on the size of my system, especially since the frame buffer and texture buffer each were about 8 megabytes in size. I overcame this by designing virtual memory systems to handle the swapping of regions of the buffer to and from the hard drive. Also, without any graphics libraries, it was impossible for me to display the frame buffer directly in the simulator. Instead, I wrote a program to convert the virtual frame buffer on the hard drive into a TIFF file that can be viewed by many programs.
- "Completeness" of architecture: My original conception of the architecture is diagrammed below. In my opinion, this would constitute a "complete" rendering architecture, because it has a functioning interface with the host, a shared frame buffer and texture memory that is accessed through a prioritized memory controller, as well as an optional external SDRAM interface, access to these memories by the host through the controller, texture caches to minimize slower DRAM reads, and a standard video output port. Unfortunately, implementing this entire system was not realistic, and I settled for the core rendering components.



Individual Contributions:

I wrote all the code, all of which is written from scratch, without any outside code. The total project consists of over 10,000 lines of C++ code, roughly half of which I consider part of the "graphics architecture". The rest is the simulator as well as standard modules for digital logic (bit-wise operations, integer arithmetic, flow-control, registers, memories, etc).

Lessons Learned:

One thing I learned was that designing the hardware description language took a lot more effort and time than I had expected. Still, the "syntax" of the language is not as nice as I had hoped it would be, and I found that designing hardware with it was a bit cumbersome and verbose. I learned a lot about what sort of desirable structures and syntaxes are desirable in the language, and if I were to remake the language, it would be a lot more fluid. However, even though making the language took a lot of work, I believe it was essential for this project. Without it, I doubt that I would have been able to run a simulation of that size efficiently. Also, the extra flexibility added by the C++ compiler proved to be very useful in designing the generic blocks (imagine having to create an adder for every bit width you needed).

One regret I have was that I designed the system at such a low level. Although it's more satisfying to have the simulation run over the full scale of hierarchical abstractions, it adds a lot of overhead to the simulation, and really isn't that instructive in terms of graphics hardware architectures. Many of the modules I created were later abstracted out in order to consume less memory or execute faster. I believe a top-down approach would have been better for this project, although it is difficult to design in hardware at such a high level when you don't know all the details that are only apparent at lower levels.

Another concept that this project teaches is the difficulty in translating simple algorithms in software into designs in hardware. Although certain tasks are easier to perform directly in hardware without the constraints imposed by the software instruction set and data types, *algorithms* always seem to be easier to formulate in software. For example, a direct software implementation that produces the same output as my hardware design could have probably be written in 100-200 lines of code in a couple of hours. Compare this with my hardware implementation of over 10,000 lines and weeks of programming. Simple programming structures such as loops and arrays can become quite complicated in hardware when you have to explicitly deal with clocking times and datapaths.

In general, I have to say that all aspects of the project were more difficult that I had originally thought. I had certainly not planned on spending so many sleepless nights on this project. The only "easy" part was perhaps implementing the lowest-level gates. Putting it all together and getting it work in the end was, in fact, perhaps one of the most challenging design problems I have ever encountered.

Acknowledgements:

I would like to thank Professor Teller and my TA Charles Lee for advising me to simplify my project and to try to remain realistic. If I hadn't at least to some degree followed that advice, it would have been impossible to complete the project in time. Also, working at SGI over the summer, I saw some of their graphics chip simulation models, which provided inspiration for creating my own. Note, however, that my system is not at *all* similar to theirs. In fact, in terms of raw output, it outperforms theirs by quite a bit ☺.

Bibliography:

Specifications for the BitBoyz Glaze3D chip, a quad-pixel integrated graphics accelerator. www.bitboyz.fi.

Their spec provided me with motivation to make a quad-pixel pipeline, and also made me aware of eDRAM technology.

Infineon Technologies CMOS ASIC information.http://www.infineon.com/Technology/ASIC2/index1.htm.

Information on 0.18micron ASICs, including densities, operating conditions, gates, eDRAM.

Appendix:

A note about the program: Since the purpose of the project was to simulate a graphics accelerator, I didn't make any form of usable user interface or interaction technique. Since the program itself is only immediately useful to myself, for a more in-depth understanding of the functioning of the program, please ask me for a "tour" of the code! Also, although the program is written to be mostly platform-independent, I have not attempted to run it on Athena or any other non-Wintel, non-Borland C++ system. It takes as input two files, the display list and the texture ROM. Its output is 8 files that represent segments of the frame buffer RAM after the display list has run to completion. A separate program then converts these files into a readable TIFF graphics file.