Aaron Lamoreaux, Julia Benginow

Verilog GPU

Design

Our main goal was to implement the graphics pipeline such that by simulating our GPU, we can render simple triangle meshes including the ability to write custom ‘shaders’ for lighting, transformations, etc. In the end, we do have a working demo which renders the classic cute little teapot, but can be given any triangle mesh as input.

We wanted our GPU to be programmable, so we created an ISA allowing the user to do this. The user can write 4 separate programs to run in the pipeline which we used for transforming vertices, calculating lighting, projection, and rasterization. We intended rasterization to be implementable in this ISA, but realized this is quite challenging, so as of now it is limited to being simulated in C++.

We borrowed the ISA and architecture design from a UW hardware design course project[1]. The ISA individually supports multiple arithmetic operations on 16 32-bit registers, general purpose memory loads and stores, comparison and branching, and ability to access work queues.

The GPU architecture consists of a pipeline of 4 separate programs which could run in parallel on multiple cores. Currently, we only have one core, however this can be easily parallelized, and if we were to work on this project in the future, then we would parallelize it. These four separate programs (transformation, lighting, shading, rasterization) are scheduled based on the sizes of our work queues. We have five work queues, which each hold a program state (just a subset of saved registers). These registers are passed from the transformation to the lighting stage, lighting to the projection stage, and so on. The last queue holds information for the z-buffer to parse, and subsequently for the z-buffer to insert into the frame buffer. You can think of this as its own core doing work dedicated just to the z-buffer and frame buffer. Aside from the cores (both the instruction core and z-buffer hardware) and work queues, we have a dedicated general purpose memory, instruction memory, and a register file.

The main geometric algorithm behind our GPU though is the rasterization process. We implemented Brasenham’s line algorithm in order to rasterize triangles without using division. We tried to write this in our ISA (as we originally wanted this to be supported), but did not have enough time, so it is only simulated in C++.

In the end, the architecture effectively renders an image, which we will show off in our presentation demo (and which you can run given our instructions below).

---

[1] https://courses.cs.washington.edu/courses/cse467/15wi/

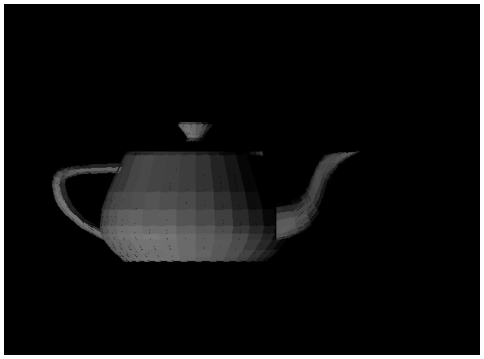
Engineering

We implemented the architecture described above in Verilog while using Verilator (in C++) in order to simulate the GPU. The simulator is implemented in *sim.cpp*. It contains the code to simulate the clock and run our Verilog implementation. Additionally it also interfaces with our memory modules to place the code into memory as well as parse the triangle meshes into memory. And it writes the final bitmap file to disk after rendering. Additionally the simulator contains code to simulate the entire GPU, but this was only used for testing except for the rasterization step which is still simulated in C++ given we had trouble implementing this in our ISA. All the assembly code we wrote is also in this file and is what is actually ran on our GPU.

The main Verilog module is implemented in *gpu.v* which constructs all the submodules for queues, memory, and the processor itself. The scheduling is done by prioritizing later queues in the pipeline although theoretically future work could make this parallel. Inside *processor.v* is the other main piece of Verilog which implements the ISA. The other modules are probably mostly self-description, e.g. *queue.v*, *regs.v*, *mem.v*, *framebuffer_mem.v*, etc.

In terms of limitations in our implementation, one rasterization is not actually implemented in our GPU even though we tried to make our ISA support this, but it's tricky to implement. Two, our Verilog is quite naive when it comes to memory hazards given that memory takes a fixed number of cycles and we use this assumption. This means further engineering work would be needed to make this run on an actual FPGA. Three, our ISA is quite constrained and could be expanded. It also has issues with signed values (especially with multiplication) which can lead to some bugs when writing shaders. Four, our rasterization and rendering as a whole does have some artifacts (for example some pixels misrendered on the edges of triangles.

In order to run our code, there is a make file and running *make clean sim* will create a *sim* executable. You can run it with *./sim object.off output.bmp* to render the object file into a bitmap image. We used the object files in objects for testing. Theoretically you can also make changes to the assembly if you follow the ISA, but the changes can be tricky, but we believe all the instructions do work. However if you do, you can transform the object, change the perspective, or the lighting direction (or anything else possible with the limited ISA).



The result of rendering the teapot.