Graphics Subsystem Emulation

Aseem Agarwala
Antoine McNamara
February 26 2002
Goals

• Fully simulate a graphics accelerator card
• Hook it into simplescalar-alpha
• Service to research community
  – Researchers can tweak various subsystems, see how they affect performance
• Learn about graphics hardware

AMBITIOUS!
Why?

- Market leading accelerator cards like GeForce3 are proprietary: no detailed descriptions of functionings exist
- Huge variety of different ways these things piece together!

• Let’s examine this a bit deeper…
How does everything talk?
Cheap

- CPU/GPU communication over shared bus
- PCI bandwidth: 133 MB/s
UMA: SGI O2

- Memory shared between all components, so less need to send data around
Intel PC’s

- Bandwidth: 512 MB/s or 1024 MB/s
Other Details

• Which graphics library?
  – OpenGL
  – DirectX

• Programmable engines
  – Vertex shaders
  – Pixel shaders
Asynchronous Execution

- GPU and CPU have different clocks, and execute asynchronously
- Communicate through FIFO
- If FIFO fills up, GPU sends interrupt to CPU and process is GPU-bound
- Otherwise, process is CPU bound

How the heck do we emulate this? Multi-thread SimpleScalar?
How much?

- How much functionality gets put into hardware?
Alright, let’s stop complaining and start making assumptions

• Memory architectures? Punt…
  – Just copy triangles from graphics application to GPU in \( n \) cycles
  – UMA, AGP-like, but won’t account for bandwidth, latency

• Asynchronous execution? Punt…
  – We will do synchronous execution, measure CPU and GPU throughput separately
  – CPU timing equivalent to assuming CPU-bound, and vice-versa; no covariance.
More specifics…

- Use OpenGL, but only SMALL subset (enough to render 3D triangles and lines)
- Concentrate on accurate simulation of graphics pipeline architecture
- Model said pipeline after SGI Indigo, since SGI likes to publish details
System Overview

- Detail: how to pass OpenGL calls through simplescalar?
  - Add several syscalls to SimpleScalar, compile pipeline right into SimpleScalar
  - Build driver to take OpenGL calls and execute syscalls through GCC’s ASM command
  - Use Mark’s Alpha cross-compiler
The Hardware Rendering Pipeline
Rendering

• For a graphics card, rendering is the process of drawing triangles from a 3D coordinate system onto a 2D screen.

• This consists of:
  – LIGHTING
  – PROJECTING
  – CLIPPING
  – SHADING
  – RASTERIZING

• How much do we do on chip?
Viewing Transformation

- As we simulate it, our chip gets vertices for triangles or lines in 3D world coordinates, with per-vertex color information.
- It also gets a matrix that transforms an arbitrary view volume onto the “canonical” view volume:
• Next, all the triangles and lines are clipped to the canonical view volume (to avoid unnecessary calculation).

• Clipped triangles may not be triangles anymore!

![Bad](image1.png) ![Worse!!](image2.png)
Rasterization

- After projecting and clipping triangles to the viewing plane, we need to draw 2D triangles to the frame buffer.
- We first break triangles into scan lines:
  - For each line, record:
    - start position and color
    - end position and color
• Next each line must be processed and drawn to the screen.

• Colors are interpolated between end points (Gouraud Shading).
Pipeline

- Modern chips do it all.
- The SGI chip starts with lighting…
- To simplify it, we’re starting with the viewing projection:
Doing it all in hardware…

• This is A LOT of computation!
  – Modern graphics chip are huge!
  – Some rival CPUs in complexity.
• But, there is A LOT of parallelism…
  – Same basic set of steps for each triangle
  – Some calculate in parallel: SIMD, MIMD
  – Others pipeline the hell out of it…
SGI chip

- The approach the chip we are simulating takes is divided into two:
  - Front-end:
    - Handles all geometric calculations
    - SIMD (single-instruction multiple data)
  - Back-end:
    - Rasterization
    - 26 level pipeline
Front End

- Primitives are put into a FIFO by the CPU
  - Pulled out in groups and handled by...
- 8 SIMD parallel Geometry Engines (GEs)
  - Each has microcode instructions for
    - Matrix multiply
    - Clipping
    - Lighting (which we don’t implement)
    - A GE has an adder and a multiplier, which can both be used in a clock cycle (the math has ~ equal +s and *s).
- All GEs need work on the same type of primitive:
  - If we have only triangles or only lines, we get ~8x.
Back End

- After coming out the GEs, vertices are in screen coords
- 26 stage pipeline
Overview

- There are actually two Rasterizers, one generates even scan lines, the other odd, writing to interleaved memory.
The Interface

• Currently, we’re using syscalls, but we might switch to memory-mapped I/O
• The basic instructions to the chip are:
  – Define Projection Matrices
  – Define Triangle
  – Define Line
  – Start Drawing!
The Driver

• The first stages of the pipeline are handled by our driver, translating OpenGL calls into syscalls (or MMIO instructions) to the chip.

• The most basic subset we support is:
  – `glBegin(GL_LINES), glBegin(GL_POLYGON)`
    • The driver must create multiple single lines and triangles.
  – `glScalef(), glTranslatef(), glRotatef()`
    • These all affect the viewing transformation matrix.
      `gluLookAt()` calls these functions to set up arbitrary camera viewpoints and directions.
Early Results!

• Well, it draws triangles (Gouraud) …

• Items completed (mostly):
  – System call interface
  – Rasterizer back-end (takes 2D triangles)

• Still do be done:
  – SIMD front-end (Geometry Engines)
  – Driver
  – Extras: Memory mapped IO, Lighting, who knows…
Conclusion

• So far, we’ve learned lots about graphics subsystems and hardware rendering.
• We provide a general framework that can be extended to accurately simulate a graphics card.
• Possible uses:
  – Tweak parameters to see performance gains with different numbers of GEs or pipeline stages.
  – Extend to support lighting
  – Extend to model asynchronous execution (hard!)
  – Reorganize to simulate more modern cards, like the nVidia GeForce 4, as specs become available.