Today’s lecture: Graphics Hardware

This W, R, F: Final project presentations (schedule is posted)
Thursday lecture: Image-based Modeling, Rendering
Friday 5pm: deadline for final project reports
Next Tuesday: best final projects presented in class

Specialized Graphics Hardware

Overview
- Bandwidth estimates for interactive rendering
- Pipelining & parallelism
- UNC Pixel Planes 5
  - Fuchs et al, Siggraph ’89
- UNC PixelFlow
  - Molnar et al, Siggraph ’92
  - Commercialized by Division
- SGI Reality Engine
  - Akeley, Siggraph ’93
- Modern Architectures
  - NVidia, ATI cards

Rendering Computation

Classical pipeline:
- Traverse database
- Transform, shade, clip
- Rasterize (with texture mapping)
- Conditional write to frame buffer

Bottlenecks:
- Memory, I/O bandwidth
- Floating point speed
- Integer (fixed point) speed
- Texture lookup bandwidth
- Frame buffer memory bandwidth
- Context switching (real-world)

Example

Sample scene with 100,000 triangles in 10 × 10 pixel bbox
- 50 pixels per triangle, of which 25 are visible (on average)
- Ambient and local diffuse illumination
- Gouraud interpolation (no texturing)
- 1280 × 1024 × 24 bit (rgb), 32 bit (depth)
- 30 Hz update rate
Per-Frame Geometry Calculations

Geometry Traversal
300,000 vertices, normals (6 FP load/store)
100,000 start/end tokens, (2 integer load/store)
2M loads/stores per frame

Modeling transformation
300,000 vertices, normals (25 mults + 18 adds)
7.5M mults, 5.4M adds per frame

Trivial accept/reject (gross clipping)
Dot each vertex against six planes
24 mults + 18 adds per vertex: 7.4M mults, 5.4M adds per frame

Lighting
12 mults + 5 adds per vertex: 3.6M mults, 1.5M adds per frame

Viewing transformation
8 mults + 6 adds per vertex: 2.4M mults and 1.8M adds

Clipping
highly variable
(assume all primitives within view frustum)

Division by \( w \)
3 divs per vertex, 0.9M divides

Transform to NDC
2 mults + 2 adds per vertex: 0.6M mults, 0.6M adds

Total for geometry:
30+ million loads & stores (depending on data
ow)
22M mults/divides + 15M adds per frame
At 30Hz frame rate, more than two GigaFlops!
(Not including application, system o/head, &c.)

Per-frame Rasterization Calculations

Visible pixels:
Compute \( z \) (1 add)
Read & compare depth (1 load, 1 compare)
Compute \( r, g, b \) (3 adds)
Write \( z, r, g, b \) to framebuffer (4 stores)

Invisible pixels:
Compute \( z \) (1 add)
Read & compare depth (1 load, 1 compare)
Each triangle covers 50 pixels, on average
100,000 \( \times \) 50 \( \times \) 5 = 15M framebuffer accesses per frame

Double-buffer clear:
1.25Mpix \( r, g, b, z \) buffer accesses = 5Mword
At 30Hz, this is 600 Million frame buffer accesses per second!
(without texture mapping!)

How do we do 2 GFlops + 600 M framebuffer ops per second?

Pipelining and Parallelism

Pipelining
Simultaneous, cascaded use of (typically) distinct computational units
Example: Washing, Drying, Folding laundry

Issues: startup, stalling, context switch

Parallelism
Simultaneous use of (typically) identical computational units

Issues: variance, inherent data parallelism

Which is well-suited to polygon rendering computation:
Pipelining or Parallelism? Why?
Graphics Pipelining and Parallelism

Pipelining:
... $B_3, B_2, B_1, A_3, A_2, A_1 \rightarrow$ Transform, Clip, Light, Rasterize

Parallelism:
- Across model (vertices can be transformed, lit in parallel)
- Across viewport (triangles can be rasterized in parallel)
- Across each triangle (pixels can be generated in parallel)

The most successful architectures have employed both organizations


Tile screen space with geometry, rasterization processors
Assemble sub-images with dedicated framebuffer hardware

Unc PixelFlow (1992)

Compositing, “deferred shading” architecture
1-128 SIMD Geometry/Rasterizer pairs
Each computes a full-screen “image” of a fraction of primitives!

Simple programming model:
- SIMD linear expression evaluators for $128 \times 128$ pixel regions
  evaluate $Ax + By + C$ (for sidedness, color) at each pixel in parallel

Backplane Architecture

Renderer emits visible pixels & depth, screen-space lighting equations
Shader emits textured, lit pixel values & depth
Framebuffer board clears, maintains and displays consistent framebuffer values
Pipeline is “factored” by visibility:
Renderer emits visible pixels & depth, screen-space lighting equations
Shader emits textured, lit pixel values & depth
Framebuffer board clears, maintains and displays consistent framebuffer values

Discussion
Every pixel must be transmitted every frame
Restrictive visibility/transparency algorithms
One full frame of storage required per renderer
Composition network causes latency
How scalable is this architecture?

Composition network bandwidth
30 GigaBits / second
Geometry processor (i860XP)
150,000 triangles / second
Rasterizer
1.4 million Gouraud triangles / second
0.8 million Phong, texture-mapped
Divide by supersampling rate (in published paper, 5)
Deferred Shader
10,000 128 × 128 regions per second
Mip-map textures: 3,700 regions per second
Remember: one primitive per region!
Bottlenecks
Linear scaling until composition time dominates
Need large datasets to scale linearly
up to 128 renderers: 400,000 triangles/sec!

Where’s the bottleneck?
**SGI Reality Engine (1993)**

Raster Manager board overview:

- **Pipelined MIMD architecture**
  - 8 MIMD geometry engines
  - 320 MIMD rasterizers

**Geometry Engines**

Host triangulates; each immediate-mode triangle has
Vertices (OS), colors, normal (OS), texture queued to CP’s input FIFO

Packet passed to geometry engine (round-robin scheduling)
- Vertices → ES & lighted; → CS & clipped; then → NDC
- Texture coordinates → NDC
- r, g, b, α, z, & texture info computed
- Triangle edge equations computed

**Reality Engine: Triangle Bus**

Why is triangle bus needed?

- Triangles are broadcast to all FG’s
- All FG’s work in parallel
- ... So what prevents load imbalance on small triangles?

**Fragment Generators**

Triangle now ready for rasterization

Broadcast on “triangle bus” to
(5, 10 or 20) “fragment generators” (process triangles in parallel — how?)
FG’s output “fragments:” pixels with color, depth, texture info

How many FG’s process tiny triangles? Large triangles?
Finely (5-, 10-, or 20-way) interleaved for balancing
Image Engines

Fragments communicated to “image engines” (16 per FG)
Local DRAM’s, each holding a subset of framebuffer
Each IE does z compare, alpha combination, FB write

REAL Performance (1993)

Host: 1-8 MIPS CPUs (25-50MHz)
6-12 GE’s, each a 50MHz i860XP (100 MFlops)
with 2Mb of local memory
1-4 RM boards, each with:
5 FGs (each with texture memory!)
80 IEs (deliver video at 500 Mb/second), and
1280 x 1024 x 256 bits/pixel framebuffer
ASIC for input, output FIFO, registers
One million triangles per second (transform)
140 million fragments per second (fill)

Reality Engine: Other capabilities

Multi-sampling (how?)
Round-robin geometry engine scheduling
Can RR be strict, or is broadcast sometimes required?
Pineda-style parallel rasterization (how?)
Scissoring
What is it? Implications?
Deep FIFO queues
Implications?
Cost?
How scaleable?

Modern Hardware

Examples: NVidia GeForce3, ATI Radeon 8500
Programmable vertex and fragment stages
Example vertex program (GeForce3), access to eye, lights, time variable:
NVidia’s 128 4-way SIMD FP instructions
Procedural deformations, Keyframe animation interpolation
Morphing, Fog effects, Lens effects, Water refraction
Example fragment program (access to eye, normal, environment map, etc.)
NVidia register combiners, DirectX pixel shaders
Per-pixel Phong shading; per-pixel reflections
Bump mapping; procedural textures
Many current limitations:
Limited instruction function
(Example: texture fetches and blends)
Limited # of instructions & registers
No looping or branching
How to simulate looping
Multi-pass with stenciled pixel disable
Ray Tracing with Programmable Hardware

Purcell et al., Siggraph 2002
Idea: re-express ray tracing as stream computation
Remove recursion, by attaching pixel coords, weight to each ray
Use a fixed-grid spatial index for traversal simplicity
Write four fragment programs:
  1. Eye-ray generator (generates rays)
  2. Traverser (generates rays in transit, or ray-voxel pairs)
  3. Intersector (generates ray-triangle hits)
      Triangle vertex data stored in texture memory
      (Proof of concept, for scenes made up solely of triangles)
  4. Shader (generates weighted color values)
Demonstrated working ray caster, ray tracer on NVidia simulator
   Expect to see much more of this kind of thing in the future