Graphics Hardware – High Level

- I/O
  - North-bridge
  - Main Memory
  - AGP 8X

- CPU
  - Host Interface
  - 3D pipeline
  - Memory Controller
  - Other Important Graphics Stuff

- R300, etc.

- gfx memory

- 256
3D Pipeline Overview

- 350 M Vertices (=Tris)/sec
- 2.6 G Pixels/sec
- Programmable vertex, fragment processors
- Allows
  - Sophisticated shading
  - High scene, depth complexity
  - Multipass
  - Computation on GPUs
New Features

• Floating-point fragments/pixels
• Usable programmable vertex, fragment engines
  – Limited; not like a CPU
• Other Features
  – Early/Hierarchical z
  – Occlusion testing
  – Antialiasing
    • Multisample (vs. downsampling)
New Applications

- Programmability
  - Complex lighting
  - Motion blur, depth of field
  - Other computations

- Floating-point pixels, textures
  - High dynamic range rendering
  - Volume rendering
  - Multipass computation
    - Many recent examples
ATI Hardware

- 9700/9800, 9800XT (R300/R350)
- 4 vertex processors
- 8 fragment processors
  - Low cost versions with 4
- 256 vertex instructions
  - Loops, subroutines may increase this
- 160 fragment instructions
  - 32 texture ops, 64 color ops, 64 alpha ops
- Up to 4 render targets
3D pipeline – vertex fetch

Getting vertex data

- Command Processor
- Vertex Processor
- Rasterizer
- Fragment Processor
- Per-Pixel Operations
- Framebuffer

Immediate mode; Direct specification

CPU

Host Interface

Memory Controller

Graphics Memory

Vertices

Vertex Fetch

Command Processor
3D pipeline – vertex fetch

Getting vertex data

Command Processor

Vertex Processor

Rasterizer

Fragment Processor

Per-Pixel Operations

Framebuffer

CPU

Host Interface

Command Processor

Memory Controller

Graphics Memory

Vertices

Vertex arrays; Indirect specification
3D pipeline – vertex fetch

Getting vertex data

- Command Processor
  - Vertex Processor
    - Rasterizer
      - Fragment Processor
        - Per-Pixel Operations
          - Framebuffer

- Graphics Memory
  - Memory Controller
    - Command Processor
      - Command Processor
        - Memory Controller
          - Commands
            - Host Interface
              - Commands
                - Vertex Fetch
                  - Vertices

- CPU

Index arrays; Indexed specification
Vertex Processors

- 4 4-component dot-product engines
  - 350 MHz
  - 4 clocks per vertex per engine
- 32-bit floating-point
- Arithmetic instructions:
  - madd, 1/x, 1/sqrt(x), etc.
  - slt (set on less than)
- Flow control instructions
  - call, return, loop, jnz, etc.
Vertex Processor - Implementation

- Command Processor
- Vertex Processor
- Rasterizer
- Fragment Processor
- Per-Pixel Operations
- Framebuffer

Diagram:

- Input Registers
  - 16x128 1W3R Register File
- Constants
  - 256x128 1W3R Register File
- Temporary Registers
  - 8x128 1W3R Register File
- ALU
- Output Registers
  - 16x128 1W1R Register File

Sequencer

Vertex Stream In

Register Interface

Vertex Stream Out
Vertex Processing – Final Steps

- Clip against view frustum
  - May introduce new vertices
  - Also “clip” parameters
  - Slow
- Perspective divide
  - Divide by w
- Viewport transform
  - Scale & offset x, y
- Triangle assembly
Scan Conversion

Convert triangles to fragments
– R300/350: 2 stage
  • Coarse: 8x8  Fine: 2x2
  • Cache friendly
  • Enables coarse depth-based rejection

![Diagram showing the process of scan conversion with stages: Command Processor, Vertex Processor, Rasterizer, Fragment Processor, Per-Pixel Operations, and Framebuffer. The diagram also illustrates Coarse Scan Conversion and Fine Scan Conversion.]
Parameter Interpolation

- Compute parameter (depth, color, texture coord) at fragment
  - Use plane equation (affine in x,y) derived from values at vertices
  - Perspective correct color, texture require division by interpolated 1/w

\[
z = a_z x + b_z y + c_z
\]
\[
s = \frac{(a_s x + b_s y + c_s)}{(a_{1/w} x + b_{1/w} y + c_{1/w})}
\]
Fragment Processors

- 8 4-component dot-product engines
  - 350 MHz
- 4x Interleaved ALU op/texture fetch
  - ALU ops:
    - MUL, ADD, EXP, etc.
    - DP3, DP4
    - 3-component + scalar coissue
  - Texture fetch
    - TEXLD, TEXLDP (projective), TEXLDBIAS
  - Interleaving means texture fetch can be free
- 24-bit floating-point
  - 32-bit components from texture or interpolation
  - No mipmapping on floating-point textures
Fragment Processor Implementation
Per-Pixel Operations

- **Fog**: \( c = c_{\text{current}} \times (1-f) + c_{\text{fog}} \times f \)
- **Alpha function**: – kill a fragment based on alpha value
- **Depth and Stencil Tests** – Conditionally kill fragment based on depth or stencil test; conditionally update depth & stencil
- **Color blend** – Blend incoming/existing colors
Framebuffer

- Floating-point framebuffer
  - 32-bit components
  - Also fixed point, including 10:10:10:2
- 340 MHz 256-bit DDR-2
  - > 20 GB/sec
- Color, texture, depth, stencil caches
  - Reduce memory bandwidth
  - Benefit from spatial, temporal locality
- Lossless color & depth compression
F-buffer

- Fragments written in rasterization order
  - x,y coordinates not used
- Useful for multipass transparency
  - no double hits
  - same geometry generates same fragments in same order
  - bind as texture in subsequent passes
Non-Features

- Data-dependent conditionals
  - Hard in a SIMD environment
    - Different processors execute different code
- Texture in vertex processors
  - Extra texture fetch, cache, memory requirements
- More vertex or fragment instructions
  - More on-chip instruction memory
- Global accumulate
  - SIMD again
- Not a CPU!
  - Performance
Graphics Programming Interfaces
Graphics Programming Interfaces

• Provide software interface to graphics hardware
• Lowest level:
  – Expose full functionality of hardware at full performance
  – Hide device-specific details
  – Limit interface changes generation to generation
• Higher levels:
  – Simplify application programming
    • E.g. scene graph, shading languages
Interface Levels

• Low level
  – Close to hardware
  – Like assembly language

• High Level
  – Insulates programmer from details
  – Shading Languages
  – Scene Graph Libraries
Low Level Interface

- Expose pipeline

  ![Image of pipeline]

  Historically fixed-function

- OpenGL
  - Requires all functionality; software path if not

- Direct3D
  - Uses capability bits

- Both provide vertex, fragment “assembly language”
High Level Interface

• Programmable
• Renderman
  – Shading-specific; not designed for hardware
• HLSL (DirectX), GLSL (OpenGL), Cg (nVidia - both)
  – General, but all expose vertices vs. fragments
  – GLSL virtualizes number of passes
• Ashli (ATI)
  – Tool to “compile” Renderman to D3D/OpenGL
High Level Issues

• How much low level to expose?
  – 3x1, 4x1, 4x4 vectors/matrices
  – Component swizzling

• Who owns the compiler?
  – Cg, HLSL, Ashli: compile to abstract machine language
  – GLSL: Compiler in the driver; target language is hidden
  – Issues for portability, performance, debugging
Re-examine hw abstraction

- OpenGL (originally)
  - Not programmable
  - Graphics memory not accessible, special purpose
- Changes:
  - Programmability
  - Flexible use of graphics memory (multipass, render-to-texture, render-to-vertex-array)
Superbuffers

• Allows application control over graphics memory allocation and use
• Render-to-texture, render-to-vertex-array
• Faster, more flexible than pbuffers
• Programming model:
  – Allocate formatted graphics memory
  – Bind to one or more compatible targets
• Preliminary implementation available
New low-level interface

- Expose Programmability; jettison fixed function
  - ARB_fragment, vertex_program
  - Use libraries
- Expose memory capabilities and routing
- Stripped-down interface
Compare with OpenGL

• No Begin/End or immediate mode
• No vertex transform
• No texture environment
• OpenGL is an application layered on this
• Benefit: simplified driver
  – Much less state management
  – No software path
  – Better support, faster addition of new features
Fits with New Applications

- Recent applications exploit high-speed, parallel computation, large graphics memory
  - Ray-tracing, collision detection, volume rendering/classification, etc.
  - Optimization, matrix computations
- All need programmability, graphics memory manipulation
- Standard polygon rendering works too
Issues

• ARB_fragment, vertex_program don’t correspond to machine instructions
  – Standard instruction set (like x86) unlikely soon
  – Use C?

• Acceptance
  – Trend is towards higher-level, shading
  – OpenGL, DirectX carry lots of baggage
What’s Next

• Graphics pipeline stays
  vertices -> triangles -> rasterize -> fragments -> pixels
• Higher clock rates
  – Memory not increasing as fast
• More parallelism
  – Probably SIMD, as today
• Cleanup & extensions
  – 32 bit floating-point, etc.
• More computation on GPUs
  – When it makes sense