GPU Memory Model Overview

Aaron Lefohn
University of California, Davis
With updates from slides by
Suresh Venkatasubramanian,
University of Pennsylvania
Updates performed by Joseph Kider,
University of Pennsylvania

Note: These slides do not include the NVIDIA 8-series memory model
Review

3D API: OpenGL or Direct3D

GPU Front End

Programmable Vertex Processor

Primitive Assembly

Assembled Primitives

Rasterization and Interpolation

Transformed Vertices

Programmable Fragment Processor

Raster Operations

Transformed Fragments

Frame Buffer

CPU-GPU Boundary (AGP/PCIe)

Fixed-function pipeline

3D Application Or Game

Vertex Index Stream

Pixel Location Stream

3D API Commands

Data Stream

GPU Command & Data Stream

Pre-transformed Vertices

GPU Front End

GPGPU

3D Application Or Game

Assembled Primitives

Rasterization and Interpolation

Transformed Vertices

Programmable Fragment Processor

Frame Buffer

Pre-transformed Fragments
Review

- Color Buffers
  - Front-left
  - Front-right
  - Back-left
  - Back-right
- Depth Buffer (z-buffer)
- Stencil Buffer
- Accumulation Buffer
Overview

• GPU Memory Model
• GPU Data Structure Basics
• Introduction to Framebuffer Objects
• Fragment Pipeline
• Vertex Pipeline
Memory Hierarchy

- CPU and GPU Memory Hierarchy

- Disk
  - CPU Main Memory
    - GPU Video Memory
      - GPU Caches
        - GPU Constant Registers
        - GPU Temporary Registers
    - CPU Caches
      - CPU Registers
CPU Memory Model

- At any program point
  - Allocate/free local or global memory
  - Random memory access
    - Registers
      - Read/write
    - Local memory
      - Read/write to stack
    - Global memory
      - Read/write to heap
  - Disk
    - Read/write to disk
GPU Memory Model

- **Much more restricted memory access**
  - Allocate/free memory only before computation
  - Limited memory access during computation (kernel)

- Registers
  - Read/write

- Local memory
  - Read/write

- Shared Memory
  - Only available in GPGPU not Graphics pipeline

- Global memory
  - Read-only during computation
  - Write-only at end of computation (pre-computed address)

- Virtual Memory
  - Does not exist

- Disk access
  - Does not exist
GPU Memory Model

• Where is GPU Data Stored?
  - Vertex buffer
  - Frame buffer
  - Texture
GPU Memory API

- Each GPU memory type supports subset of the following operations
  - CPU interface
  - GPU interface
GPU Memory API

• CPU interface
  - Allocate
  - Free
  - Copy CPU → GPU
  - Copy GPU → CPU
  - Copy GPU → GPU
  - Bind for read-only vertex stream access
  - Bind for read-only random access
  - Bind for write-only framebuffer access
GPU Memory API

- GPU (shader/kernel) interface
  - Random-access read
  - Stream read
Vertex Buffers

- GPU memory for vertex data
- Vertex data required to initiate render pass
Vertex Buffers

• Supported Operations
  - CPU interface
    • Allocate
    • Free
    • Copy CPU → GPU
    • Copy GPU → GPU (Render-to-vertex-array)
    • Bind for read-only vertex stream access
  - GPU interface
    • Stream read (vertex program only)
Vertex Buffers

• Limitations
  - CPU
    • No copy GPU → CPU
    • No bind for read-only random access
    • No bind for write-only framebuffer access
  - GPU
    • No random-access reads
    • No access from fragment programs
Textures

- Random-access GPU memory
Textures

• Supported Operations

  - CPU interface
    • Allocate
    • Free
    • Copy CPU → GPU
    • Copy GPU → CPU
    • Copy GPU → GPU (Render-to-texture)
    • Bind for read-only random access (vertex or fragment)
    • Bind for write-only framebuffer access

  - GPU interface
    • Random read
Textures

• Limitations
  - No bind for vertex stream access
Framebuffer

- Memory written by fragment processor
- Write-only GPU memory
OpenGL Framebuffer Objects

- General idea
  - Framebuffer object is lightweight struct of pointers
  - Bind GPU memory to framebuffer as write-only
  - Memory cannot be read while bound to framebuffer

- Which memory?
  - Texture
  - Renderbuffer
  - Vertex buffer??
What is a Renderbuffer?

- “Traditional” framebuffer memory
  - Write-only GPU memory
    - Color buffer
    - Depth buffer
    - Stencil buffer

- New OpenGL memory object
  - Part of Framebuffer Object extension
Renderbuffer

• Supported Operations
  - CPU interface
    • Allocate
    • Free
    • Copy GPU → CPU
    • Bind for write-only framebuffer access
Pixel Buffer Objects

• Mechanism to efficiently transfer pixel data
  - API nearly identical to vertex buffer objects
Pixel Buffer Objects

- **Uses**
  - Render-to-vertex-array
    - `glReadPixels` into GPU-based pixel buffer
    - Use pixel buffer as vertex buffer
  - Fast streaming textures
    - Map PBO into CPU memory space
    - Write directly to PBO
    - Reduces one or more copies
Pixel Buffer Objects

• Uses (continued)
  - Asynchronous readback
    • Non-blocking GPU → CPU data copy
    • glReadPixels into PBO does not block
    • Blocks when PBO is mapped into CPU memory
Summary: Render-to-Texture

- Basic operation in GPGPU apps

- OpenGL Support
  - Save up to 16, 32-bit floating values per pixel
  - Multiple Render Targets (MRTs) on ATI and NVIDIA

1. Copy-to-texture
   - `glCopyTexImage2D`

Render-to-texture
- `GL_EXT_framebuffer_object`
Summary: Render-To-Vertex-Array

• Enable top-of-pipe feedback loop

• OpenGL Support
  - Copy-to-vertex-array
    • GL_ARB_pixel_buffer_object
    • NVIDIA and ATI
  - Render-to-vertex-array
    • Maybe future extension to framebuffer objects
MRT allows us to compress multiple passes into a single one.

This does not fundamentally change the model though, since read/write access is still not allowed.
Overview

• GPU Memory Model
• GPU Data Structure Basics
• Introduction to Framebuffer Objects
• Fragment Pipeline
• Vertex Pipeline
GPU Arrays

- **Large 1D Arrays**
  - Current GPUs limit 1D array sizes to 2048 or 4096
  - Pack into 2D memory
  - 1D-to-2D address translation
GPU Arrays

3D Arrays

- Problem
  - GPUs do not have 3D frame buffers
  - No render-to-slice-of-3D-texture yet (coming soon?)

- Solutions
  1. Stack of 2D slices
  2. Multiple slices per 2D buffer
GPU Arrays

Problems With 3D Arrays for GPGPU
- Cannot read stack of 2D slices as 3D texture
- Must know which slices are needed in advance
- Visualization of 3D data difficult

Solutions
- Flat 3D textures
- Need render-to-slice-of-3D-texture
  - Maybe with GL_EXT_framebuffer_object
- Volume rendering of flattened 3D data
  - “Deferred Filtering: Rendering from Difficult Data Formats,”
    GPU Gems 2, Ch. 41, p. 667
GPU Arrays

• Higher Dimensional Arrays
  - Pack into 2D buffers
  - N-D to 2D address translation
  - Same problems as 3D arrays if data does not fit in a single 2D texture
Sparse/Adaptive Data Structures

• Why?
  - Reduce memory pressure
  - Reduce computational workload

• Examples
  - Sparse matrices
    • Krueger et al., Siggraph 2003
    • Bolz et al., Siggraph 2003
  - Deformable implicit surfaces (sparse volumes/PDEs)
    • Lefohn et al., IEEE Visualization 2003 / TVCG 2004
  - Adaptive radiosity solution (Coombe et al.)
Sparse/Adaptive Data Structures

• Basic Idea
  - Pack “active” data elements into GPU memory
Overview

• GPU Memory Model
• GPU-Based Data Structures
• Introduction to Framebuffer Objects
• Fragment Pipeline
• Vertex Pipeline
Framebuffer Objects

- What is an FBO?
  - A struct that holds pointers to memory objects
  - Each bound memory object can be a framebuffer rendering surface
  - Platform-independent
Framebuffer Objects

• Which memory can be bound to an FBO?
  - Textures
  - Renderbuffers
    • Depth, stencil, color
    • Traditional write-only framebuffer surfaces
Framebuffer Objects

• Usage models
  - Keep N textures bound to one FBO (up to 16)
    • Change render targets with glDrawBuffers
  - Keep one FBO for each size/format
    • Change render targets with attach/unattach textures
  - Keep several FBOs with textures attached
    • Change render targets by binding FBO
Framebuffer Objects

• Performance
  - Render-to-texture
    • `glDrawBuffers` is fastest on NVIDIA/ATI
      - As-fast or faster than `pbuffers`
  - Attach/unattach textures same as changing FBOs
    • Slightly slower than `glDrawBuffers` but faster than `wglMakeCurrent`
  • Keep format/size identical for all attached memory
    - Current driver limitation, not part of spec

• Readback
  • Same as `pbuffers` for NVIDIA and ATI
Framebuffer Objects

• Driver support still evolving
  - GPUBench FBO tests coming soon...
  • “fbocheck” evaluates completeness
  • Other tests...
Framebuffer Object

- Code examples
  - Simple C++ FBO and Renderbuffer classes
    - HelloWorld example

- OpenGL Spec
Overview

- GPU Memory Model
- GPU Data Structure Basics
- Introduction to Framebuffer Objects
- Fragment Pipeline
- Vertex Pipeline
Review

3D API: OpenGL or Direct3D

GPU Front End

Command & Data Stream

GPU

Pre-transformed Vertices

Programmable Vertex Processor

3D Application Or Game

3D API Commands

Vertex Index Stream

Assembled Primitives

Rasterization and Interpolation

Transformed Vertices

Pre-transformed Fragments

Programmable Fragment Processor

Frame Buffer

Raster Operations

Transformed Fragments

Pixel Updates

CPU-GPU Boundary (AGP/PCIe)

Fixed-function pipeline

3D Application Or Game

3D API: OpenGL or Direct3D

GPU Front End

Programmable Vertex Processor

3D API Commands

CPU-GPU Boundary (AGP/PCIe)

Fixed-function pipeline
The fragment pipeline

Input: Fragment

Attributes

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>G</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Color</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Position</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>W</td>
</tr>
<tr>
<td>Texture coordinates</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>-</td>
</tr>
<tr>
<td>Texture coordinates</td>
<td>X</td>
<td>Y</td>
<td>Z</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32 bits = float
16 bits = half

Input: Texture Image

- Each element of texture is 4D vector
- Textures can be “square” or rectangular (power-of-two or not)
The fragment pipeline

Input: Uniform parameters
• Can be passed to a fragment program like normal parameters
• set in advance before the fragment program executes

Example:
A counter that tracks which pass the algorithm is in.

Input: Constant parameters
• Fixed inside program
• E.g. float4 v = (1.0, 1.0, 1.0, 1.0)

Examples:
3.14159..
Size of compute window
The fragment pipeline

Math ops: USE THEM!
- cos(x)/log2(x)/pow(x,y)
- dot(a,b)
- mul(v, M)
- sqrt(x)
- cross(u, v)

Using built-in ops is more efficient than writing your own

Swizzling/masking: an easy way to move data around.

\[
\begin{align*}
v1 &= (4,-2,5,3); \quad // \text{Initialize} \\
v2 &= v1.yx; \quad // v2 = (-2,4) \\
s &= v1.w; \quad // s = 3 \\
v3 &= s.rrr; \quad // v3 = (3,3,3)
\end{align*}
\]

Write masking:

\[
\begin{align*}
v4 &= (1,5,3,2); \\
v4.ar &= v2; \quad // v4=(4,5,4,-2)
\end{align*}
\]
The fragment pipeline

float4 v = tex2D(IMG, float2(x,y))

Texture access is like an array lookup.
The value in v can be used to perform another lookup!
This is called a dependent read

Texture reads (and dependent reads) are expensive resources, and are limited in different GPUs. Use them wisely!
The fragment pipeline

Control flow:

• $(<\text{test}>)?a:b$ operator.
• if-then-else conditional
  - [nv3x] Both branches are executed, and the condition code is used to decide which value is used to write the output register.
  - [nv40] True conditionals
• for-loops and do-while
  - [nv3x] limited to what can be unrolled (i.e. no variable loop limits)
  - [nv40] True looping.

WARNING: Even though nv40 has true flow control, performance will suffer if there is no coherence (more on this later)
The fragment pipeline

Fragment programs use **call-by-result**

Notes:

- Only output color can be modified
- Textures **cannot be written**
- Setting different values in different channels of result can be useful for debugging

```plaintext
out float4 result : COLOR
// Do computation
result = <final answer>
```
Overview

- GPU Memory Model
- GPU Data Structure Basics
- Introduction to Framebuffer Objects
- Fragment Pipeline
- Vertex Pipeline
Review

3D API: OpenGL or Direct3D

3D Application Or Game

CPU-GPU Boundary (AGP/PCIe)

GPU Front End

Primitive Assembly

Rasterization and Interpolation

Raster Operations

Frame Buffer

GPU Command & Data Stream

Fixed-function pipeline

3D API Commands

Vertex Index Stream

Assembled Primitives

Pixel Location Stream

Pixel Updates

Transformed Vertices

Pre-transformed Vertices

Pre-transformed Fragments

Transformed Fragments

Programmable Vertex Processor

Programmable Fragment Processor

3D Application Or Game

Assembled Primitives

Rasterization and Interpolation

Raster Operations

Frame Buffer
The Vertex Pipeline

Input: vertices
• position, color, texture coords.

Input: uniform and constant parameters.
• Matrices can be passed to a vertex program.
• Lighting/material parameters can also be passed.
The Vertex Pipeline

Operations:
- Math/swizzle ops
- Matrix operators
- Flow control (as before)

[nv3x] No access to textures.

Output:
- Modified vertices (position, color)
- Vertex data transmitted to primitive assembly.
Vertex programs are useful

- We can replace the entire geometry transformation portion of the fixed-function pipeline.
- Vertex programs used to change vertex coordinates (move objects around)
- There are many fewer vertices than fragments: shifting operations to vertex programs improves overall pipeline performance.
- Much of shader processing happens at vertex level.
- We have access to original scene geometry.
Vertex programs are not useful for GPGPU

- Fragment programs allow us to exploit full parallelism of GPU pipeline ("a processor at every pixel").
- Vertex programs can’t read input! [nv3x]
- NV4X Cards can read vertex textures but can not read FBOs

Rule of thumb:
If computation requires intensive calculation, it should probably be in the fragment processor.
If it requires more geometric/graphic computing, it should be in the vertex processor.
Acknowledgements

- Adam Moerschell, Shubho Sengupta, UCDavis
- Mike Houston, Stanford University
- John Owens, Ph.D. advisor, UC Davis

- National Science Foundation Graduate Fellowship

- Extra slides were added by Joe Kider, Gary Katz from Suresh Venkatasubramanian, lecture 3 found at http://www.cis.upenn.edu/~suvenkat/700/

- Alteration to this slide package were made without the authorization by the original authors and should be used for educational purposes only.