Lecture 3: 
Parallelizing Pipeline Execution 
(+ notes on workload)

Kayvon Fatahalian 
CMU 15-869: Graphics and Imaging Architectures (Fall 2011)
Today

- Brief discussion of graphics workload
- Strategies for parallelizing the graphics pipeline
The graphics pipeline (last time)

Vertices
- 1 in / 1 out
- 3 in / 1 out (for tris)

Primitives
- 1 in / small N out

Fragments
- 1 in / N out
- Rasterization (Fragment Generation)

Frames
- 1 in / 1 out

Pixels
- 1 in / 0 or 1 out

Memory
- Uniform data
- Texture buffers
- Frame Buffer
Programming the pipeline (last time)

- Issue draw commands → frame-buffer contents change

<table>
<thead>
<tr>
<th>Command Type</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>State change</td>
<td>Bind shaders, textures, uniforms</td>
</tr>
<tr>
<td>Draw</td>
<td>Draw using vertex buffer for object 1</td>
</tr>
<tr>
<td>State change</td>
<td>Bind new uniforms</td>
</tr>
<tr>
<td>Draw</td>
<td>Draw using vertex buffer for object 2</td>
</tr>
<tr>
<td>State change</td>
<td>Bind new shader</td>
</tr>
<tr>
<td>Draw</td>
<td>Draw using vertex buffer for object 3</td>
</tr>
<tr>
<td>State change</td>
<td>Change depth test function</td>
</tr>
<tr>
<td>State change</td>
<td>Bind new shader</td>
</tr>
<tr>
<td>Draw</td>
<td>Draw using vertex buffer for object 4</td>
</tr>
</tbody>
</table>

Note: efficiently managing stage changes is a major challenge in implementations
Where is the work?

Vertices
- 1 in / 1 out
  - Vertex Generation
  - Vertex Processing

Primitives
- 3 in / 1 out (for tris)
  - Primitive Generation
  - Primitive Processing

Fragments
- 1 in / N out
  - Rasterization (Fragment Generation)
  - Fragment Processing

Pixels
- 1 in / 0 or 1 out
  - Frame-Buffer Ops

Memory
- Uniform data
- Texture buffers

Frame Buffer

Kayvon Fatahalian, Graphics and Imaging Architectures (CMU 15-869, Fall 2011)
Triangle size

Note: tessellation is triggering a reduction in triangle size

(source: NVIDIA)
“Diamond” structure of graphics workload

Amount of data

Compact model

High-resolution mesh

screen-space fragments

Frame buffer pixels

1 in / 1 out

Coarse Vertices

1 in / 1 out

Vertex Generation

1 in / N out

Coarse Primitives

3 in / 1 out (for tris)

Fine Primitives

1 in / small N out

Tessellation

1 in / 1 out

Fine Vertices

Fine Primitive Generation

1 in / N out

Fine Primitives

1 in / 1 out

Rasterization

(Fragment Generation)

1 in / 1 out

Fragments

Fine Primitive Processing

1 in / 0 or 1 out

Frame-Buffer Ops

1 in / 1 out

Frame buffer pixels

Kayvon Fatahalian, Graphics and Imaging Architectures (CMU 15-869, Fall 2011)
Key workload metrics

- Data amplification
  - Triangle size
  - Expansion by geometry shader (if enabled)
  - Tessellation factor (if enabled)

- [Vertex/fragment] program cost

- Depth Complexity
  - Determines number of z/color buffer writes
Scene depth complexity

Loose approximation: $TA = SD$

$T = \#$ triangles
$A = \text{average triangle area}$
$S = \text{pixels on screen}$
$D = \text{average depth complexity}$
Pipeline workload changes rapidly

- Triangle size scene and frame dependent
  - Even object dependent within a frame (characters: higher res meshes)

- Varying complexity of materials, different number of lights illuminating surfaces
  - No “average” shader
  - Tens to several hundreds of instructions per shader

- Shadow map creation
  - NULL fragment shader

- Screen post-processing
  - Two triangles cover screen
    (~ no vertex work)

- Recall: thousands of draw calls per frame
Parallelization

Some slides credit Kurt Akeley and Pat Hanrahan (Stanford CS448 Spring 2007)
Remember our workload

- **Immediate mode interface:** accepts sequence of commands
  - draw commands
  - state modification commands

- **Processing of commands has sequential semantics**
  - Effects of command A visible before those of command B

- **Relative cost of pipeline stages changes frequently and unpredictably** (e.g., triangle size)

- **Ample opportunities for parallelism**
  - few dependencies (most notable: order, frame-buffer update)
Parallelism and communication

- **Parallelism**: Using multiple execution units to process work in parallel

- **Communication**: Connecting the execution units allowing work to be distributed and aggregated
  
  (note: consider synchronization a form of communication)

- **Issues**:
  - Scalability:
    - Computation
    - Bandwidth
    - Load-balancing
  - Dependencies (ordering semantics)
  - Work efficiency
Opportunities for parallelism in graphics

- Data parallelism
  - Simultaneously execute same operation on different data
  - Object space (vertices, primitives, etc.)
  - Image space (fragments, pixels)

- Task parallelism
  - Simultaneously execute different tasks on similar (or different) data
  - Vertex processing, rasterization, fragment processing

Note: many redundancies in the pipeline: optimizations exploiting these redundancies can create dependencies that reduce opportunities of parallelism
Simple parallelization (pipelined)

- Vertex Generation
- Vertex Processing
- Primitive Generation
- Primitive Processing
- Rasterization (Fragment Generation)
- Fragment Processing
- Frame-Buffer Ops

Separate hardware unit for each stage

Speedup?
Simplified pipeline

For now: just consider all geometry processing work (vertex/primitive processing, tessellation, etc.) as “geometry” processing.
Scaling “wide”

Application

- Command Processing
- Geometry Processing
- Rasterization
- Fragment Processing
- Frame-Buffer Ops

Display

- Command Processing
- Geometry Processing
- Rasterization
- Fragment Processing
- Frame-Buffer Ops

- Command Processing
- Geometry Processing
- Rasterization
- Fragment Processing
- Frame-Buffer Ops

- Command Processing
- Geometry Processing
- Rasterization
- Fragment Processing
- Frame-Buffer Ops

- Command Processing
- Geometry Processing
- Rasterization
- Fragment Processing
- Frame-Buffer Ops
Sorting taxonomy

Application

Sort first
- Command Processing
- Geometry Processing
- Rasterization
- Fragment Processing
- Frame-Buffer Ops

Sort middle
- Command Processing
- Geometry Processing
- Rasterization
- Fragment Processing
- Frame-Buffer Ops

Sort last fragment
- Command Processing
- Geometry Processing
- Rasterization
- Fragment Processing
- Frame-Buffer Ops

Sort last image composition
- Command Processing
- Geometry Processing
- Rasterization
- Fragment Processing
- Frame-Buffer Ops

Display
Sort first
Assign each hardware pipeline a region of the render target
Do minimal amount of work to determine which region(s) input primitive overlaps
Sort first

- Good:
  - Bandwidth scaling (small amount of sync/communication, simple point-to-point)
  - Computation scaling
  - Simple: just replicate rendering pipeline (order maintained within each)
  - Easy early fine occlusion cull (“early z”)
Sort first

- **Bad:**
  - Potential for workload imbalance (one part of screen contains most of scene)
  - Extra cost of “pre-transformation”
  - Tile spread: as screen tiles get smaller, primitives cover more tiles (duplicate geometry processing)
Sort first examples

- **WireGL/Chromium** (parallel rendering with a cluster of GPUs)
  - “front-end” sorts primitives
  - each GPU is a full rendering pipeline

- Pixar RenderMan (implementation of REYES)
  - Multi-core software implementation
  - Sort surfaces into tiles prior to tessellation
    (sort the surfaces, not all the little “micropolygons”)

**Chromium can also be configured as a sort-last system**
Sort middle
Assign each rasterizer a region of the render target
Distribute primitives to top of pipelines (e.g., round robin)
Sort after geometry processing based on screen space projection of primitive vertices
Interleaved mapping of screen

- Decrease chance of one rasterizer processing most of scene
- Most triangles overlap multiple screen regions

Fuchs - Interleaved

Parke - Tiled
Sort middle interleaved

Good:
- Workload balance: both for geometry work AND onto rasterizers
- Computation scaling
- Easy fine early occlusion cull
- Does not duplicate geometry processing for each overlapped screen region
Sort middle interleaved

Bad:
- Bandwidth scaling: sort implemented as a broadcast (each triangle goes to many/all rasterizers)
- If tessellation enabled, must communicate many more primitives than sort first
SGI RealityEngine

Sort-middle interleaved

System Bus
Command Processor
Geometry Engines
Triangle Bus
Fragment Generators
Image Engines

geometry board

raster memory board

raster memory board

display generator board

video
Sort middle tiled

- Sort no longer requires broadcast
  - Point-to-point communication
  - Better bandwidth scaling

- Risks workload imbalance amongst rasterizers
Partition screen into many small tiles (many more tiles than rasterizers)
Sort geometry by tile into off-chip buckets.
After all geometry complete, rasterizers process buckets (think work queue)
Sort middle tiled (chunked)

- Inserts frame of delay
  - Cannot begin rasterization until geometry processing completes (order)

- Requires off-chip storage of immediate data

- Good:
  - Sort approaches point to point traffic
  - Good load balance
  - Low bandwidth requirements (why?)

- Recent examples: Intel Larrabee, NVIDIA CUDA rasterizer, many mobile GPUs
Sort last
Distribute primitives to top of pipelines (e.g., round robin)
Sort after fragment processing based on (x,y) position of fragment
Sort last fragment

Good:
- No redundant work (geometry processing or in rast)
- Point-to-point communication during sort
- Interleaved pixel mapping results in good workload balance for frame-buffer ops
Sort last fragment

- **Bad:**
  - Workload imbalance due to primitives of varying size
  - Bandwidth scaling: many more fragments than triangles
  - Hard to implement early occlusion cull (more bandwidth challenges)
Sort last image composition

Each pipeline renders some part of the frame (color buffer + depth buffer)
Combine the color buffers, according to depth into the final image
Sort last image composition

Z comp

Other combiners possible
Sort last image composition

- Cannot maintain order

- Simple: N separate rendering pipelines
  - Can use off the shelf GPUs
  - Coarse-grained communication

- Similar load imbalance problems as sort-last fragment

- Bandwidth requirements compared to sort-last fragment depend on scene depth complexity
Sort everywhere
Distribute primitives to top of pipelines
Redistribute after geometry processing (e.g., round robin)
Sort after fragment processing based on (x,y) position of fragment
Recall: modern OpenGL 4/Direct3D 11 pipeline

5 programmable stages

Tessellation

Programmable stages with data-dependent control flow (varying per vertex/per fragment run-time)
Modern NVIDIA, AMD, Intel GPUs

Hardware is a heterogeneous collection of resources
Programmable resources are time-shared by vertex/primitive/fragment processing work
Must keep programmable cores busy: sort everywhere
Readings
