Lecture 2: Parallelizing Graphics Pipeline Execution
(+ Basics of Characterizing a Rendering Workload)

Visual Computing Systems
CMU 15-869, Fall 2014
Analyzing a 3D Graphics Workload
Where is most of the work done?

- **Vertices**: 1 in / 1 out
- **Primitives**: 3 in / 1 out (for tris)
- **Fragments**: 1 in / N out
- **Pixels**: 1 in / 0 or 1 out

**Memory Diagram**:
- **Vertex Generation**
- **Vertex Processing**
- **Primitive Generation**
  - **Primitive Processing**
  - **Rasterization (Fragment Generation)**
  - **Fragment Processing**
- **Frame-Buffer Ops**

**Memory**
- Uniform data
- Texture buffers

奇怪的数据流图，需要进一步解释。
Triangle size (data from 2010)

Note: tessellation is triggering a reduction in triangle size
Graphics pipeline with tessellation
(OpenGL 4, Direct3D 11)

Vertices
1 in / 1 out

Primitives
3 in / 1 out (for tris)
1 in / small N out

Fragments
1 in / N out

Pixels
1 in / 0 or 1 out

Vertex Generation
Vertex Processing

Primitive Generation
Primitive Processing

Rasterization
(Fragment Generation)

Fragment Processing

Fine Vertices
1 in / 1 out

Fine Primitives
3 in / N out (for tris)
1 in / small N out

Fine Vertex Processing
Fine Primitive Processing

Tessellation

Rasterization
(Fragment Generation)

Fragment Processing

Frame-Buffer Ops

Coarse Vertices
1 in / 1 out

Coarse Primitives
1 in / 1 out

Fragments
1 in / N out

Pixels
1 in / 0 or 1 out
Tessellation

- Procedurally generate fine triangle mesh from coarse mesh representation

[Image credit: NVIDIA]
<table>
<thead>
<tr>
<th>Amount of data generated (size of stream between stages)</th>
<th>Compact geometric model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>High-resolution mesh</td>
</tr>
<tr>
<td></td>
<td>“Diamond” structure of graphics workload</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Diamond Structure of Graphics Workload

- **Coarse Vertices**
  - 1 in / 1 out
  - **Vertex Processing**

- **Coarse Primitives**
  - 1 in / N out
  - **Tessellation**

- **Fine Vertices**
  - 1 in / 1 out
  - **Fine Vertex Processing**

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

- **Fine Primitives**
  - 1 in / small N out
  - **Fine Primitive Processing**

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

- **Fragments**
  - 1 in / 1 out
  - **Fragment Processing**

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

### Amount of Data Generated

- **Compact geometric model**
  - 1 in / 1 out

- **High-resolution mesh**
  - 1 in / 1 out

- **Fragments**
  - 1 in / 1 out

- **Frame buffer pixels**
  - 1 in / 0 or 1 out
Key 3D graphics workload metrics

- Data amplification from stage to stage
  - Triangle size (amplification in rasterizer)
  - Expansion by geometry shader (if enabled)
  - Tessellation factor (if tessellation enabled)

- [Vertex/fragment/geometry] shader cost
  - How many instructions?
  - Ratio of math to data access instructions?

- Scene depth complexity
  - Determines number of depth and color buffer writes
Scene depth complexity

Very rough approximation: \( TA = SD \)

- \( T \) = # triangles
- \( A \) = average triangle area
- \( S \) = pixels on screen
- \( D \) = average depth complexity
Graphics pipeline workload characteristics change dramatically across draw commands

- Triangle size is scene and frame dependent
  - Move far away from an object, triangles get smaller
  - Object dependent within a frame (characters are usually higher resolution meshes)

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

- Stages can be disabled
  - Shadow map creation = NULL fragment shader
  - Post-processing effects = no vertex work

- Thousands of draw calls per frame

Example: rendering a “depth map” requires vertex shading but no fragment shading
Basics of parallelizing the graphics pipeline

Adopted from slides by Kurt Akeley and Pat Hanrahan
(Stanford CS448 Spring 2007)
Reminder: requirements + workload challenges

- Immediate mode interface: pipeline accepts sequence of commands
  - Draw commands
  - State modification commands

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

- Relative cost of pipeline stages changes frequently and unpredictably
  (e.g., due to changing triangle size, rendering mode)

- Ample opportunities for parallelism
  - Many triangles, vertices, fragments, etc.
Parallelism and communication

- **Parallelism** - using multiple execution units to process work in parallel

- **Communication** - parallel execution units must synchronize and communicate to cooperatively perform a rendering task
  - Communication between execution units (on chip network)
  - Communication between execution units and memory

- **Big issues:**
  - Correctness (preserving sequential semantics of pipeline abstraction)
  - Achieving good workload balance (using all processors)
  - Minimizing communication/synchronization
  - Avoiding unnecessary work (or duplicating work)
Simplified pipeline

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

(I’m drawing the pipeline this way to match tonight’s readings)
Simple parallelization (pipeline parallelism)

Separate hardware unit is responsible for executing work in each stage

What is my speedup?
Alternative parallelization: scaling "wide"

Leverages data-parallelism present in rendering computation
Molnar’s sorting taxonomy

Implementations characterized by where communication occurs

Note: The term “sort” can be misleading for some.* It may be helpful to instead consider the term “distribution” rather than sort. The implementations are characterized by how and when they redistribute work onto processors.

* The origin of the term sort was from “A Characterization of Ten Hidden-Surface Algorithms”. Sutherland et al. 1974
Sort first
Sort first

Assign each replicated pipeline responsibility for computing a region of the output image.
Do minimal amount of work to determine which region(s) each input primitive overlaps.
Sort first work partitioning

(partition the primitives to parallel units based on screen overlap)
**Sort first**

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

![Diagram of rendering pipeline with applications and processing stages](attachment:image.png)
Sort first

- Potential for workload imbalance (one part of screen contains most of scene)
- Extra cost of triangle “pre-transformation” (do some vertex work twice)
- “Tile spread”: as screen tiles get smaller, primitives cover more tiles (duplicate geometry processing across the parallel pipelines)
Sort first examples

- WireGL/Chromium* (parallel rendering with a cluster of GPUs)
  - “Front-end” sorts primitives to machines
  - Each GPU is a full rendering pipeline

- Pixar’s RenderMan (implementation of REYES)
  - Multi-core software renderer
  - Sort surfaces into tiles prior to tessellation

* Chromium can also be configured as a sort-last image composition system
Sort middle
Assign each rasterizer a region of the render target
Distribute primitives to pipelines (e.g., round-robin distribution)
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 (often overlap all)
Fragment interleaving in NVIDIA Fermi

**Fine granularity interleaving**

**Coarse granularity interleaving**

**Question 1:** what are the benefits/weaknesses of each interleaving?

**Question 2:** notice anything interesting about these patterns?

[Image source: NVIDIA]
Good:
- Workload balance: both for geometry work AND onto rasterizers (due to interleaving)
- Does not duplicate geometry processing for each overlapped screen region
Sort middle interleaved

- **Bad:**
  - Bandwidth scaling: sort is implemented as a broadcast (each triangle goes to many/all rasterizers)
  - If tessellation is enabled, must communicate many more primitives than sort first
  - Duplicated per triangle setup work across rasterizers
SGI RealityEngine  [Akeley 93]
Sort-middle interleaved design

[Diagram of SGI RealityEngine with labeled components: System Bus, Command Processor, Geometry Engines, Triangle Bus, Fragment Generators, Image Engines, Raster Memory Board, Display Generator Board, Video output]
**Tiling** (a.k.a. “chunking”, “bucketing”)

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>Processor 2</th>
<th>Processor 3</th>
<th>Processor 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 0 1</td>
<td>2 3 0 1 2 3</td>
<td>0 1 2 3 0 1</td>
<td>2 3 0 1 2 3</td>
</tr>
</tbody>
</table>

**Interleaved (static) assignment to processors**

<table>
<thead>
<tr>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
</tr>
</thead>
<tbody>
<tr>
<td>B6</td>
<td>B7</td>
<td>B8</td>
<td>B9</td>
<td>B10</td>
<td>B11</td>
</tr>
<tr>
<td>B12</td>
<td>B13</td>
<td>B14</td>
<td>B15</td>
<td>B16</td>
<td>B17</td>
</tr>
<tr>
<td>B18</td>
<td>B19</td>
<td>B20</td>
<td>B21</td>
<td>B22</td>
<td>B23</td>
</tr>
</tbody>
</table>

**Assignment to buckets**

List of buckets is a work queue. Buckets are dynamically assigned to processors.
Partition screen into many small tiles (many more tiles than physical rasterizers)
Sort geometry by tile into buckets (one bucket per tile of screen)
After all geometry is bucketed, rasterizers process buckets in parallel
Sort middle tiled (chunked)

- **Good:**
  - Sort requires point-to-point traffic (assuming each triangle only touches a few buckets)
  - Good load balance (distribute many buckets onto rasterizers)
  - Potentially low bandwidth requirements (why? when?)
    - Question: What should the size of tiles be for maximum BW savings?

- **Recent examples:**
  - Mobile GPUs: Imagination PowerVR, ARM Mali, etc.
  - Parallel software rasterizers
    - Intel Larrabee software rasterizer
    - NVIDIA CUDA software rasterizer
Sort last
Sort last fragment

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 geometry processing or in rasterizers (but z-cull is a problem)
  - Point-to-point communication during sort
  - Interleaved pixel mapping results in good workload balance for frame-buffer ops
Sort last fragment

- Workload imbalance due to primitives of varying size (due to order)
- Bandwidth scaling: many more fragments than triangles
- Hard to implement early occlusion cull (more bandwidth challenges)
Each pipeline renders some fraction of the geometry in the scene. 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 pipeline’s sequential semantics

- Simple implementation: N separate rendering pipelines
  - Can use off-the-shelf GPUs to build a massive rendering system
  - Coarse-grained communication (image buffers)

- Similar load imbalance problems as sort-last fragment

- Under high depth complexity, bandwidth requirement is lower than sort last fragment
  - Communicate final pixels, not all fragments
Sort everywhere
**Pomegranate** [Eldridge 00]

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

Five programmable stages
Modern GPU: programmable parts of pipeline virtualized on pool of programmable cores

Hardware is a **heterogeneous** collection of resources (programmable and non-programmable)

Programmable resources are time-shared by vertex/primitive/fragment processing work
Must keep programmable cores busy: sort everywhere
Hardware work distributor assigns work to cores (based on contents of inter-stage queues)
Readings
