A Sorting Classification of Parallel Rendering

Molnar et al., 1994

Exploiting Parallelism

- Functional parallelism (pipelining)
- Data parallelism
  - Object-parallelism
  - Image-parallelism
Adapting the pipeline to parallel rendering

- Geometry processing
  - T&L
  - Clipping
- Rasterization
  - Scan conversion
  - Shading
  - Visibility
  - etc.

Parallel rendering as a sorting problem

- **Sort-first**
  - during geometry processing
  - distributes “raw” primitives
- **Sort-middle**
  - between geom. processing and rasterization
  - distributes screen-space primitives
- **Sort-last**
  - during rasterization
  - distributes pixels/fragments
Sort-first

- Primitives initially assigned arbitrarily
- Pre-transformation is done to determine which screen regions are covered
- Primitives are then redistributed over the network to the correct renderer
- Renderer performs the work of the entire pipeline for that primitive from that point on

Sort-first analysis

- Overhead:
  - Overlap factor
  - Pre-transform cost
  - Primitive distribution cost
    - depends on fraction of primitives being redistributed
Sort-first analysis

Pros:
- Low communication requirements when tesselation or oversampling are high, or when inter-frame coherence exploited
- Processors implement entire rendering pipeline for a given screen region

Cons:
- Susceptible to load imbalance (clumping)
- Exploiting coherence is difficult

Sort-middle

- Primitives initially assigned arbitrarily
- Primitives fully transformed, lit, etc., by the geometry processor to which they are initially assigned
- Transformed primitives are distributed over the network to the rasterizer assigned to their region of the screen
Sort-middle analysis

- Overhead:
  - Display primitive distribution cost
  - Tessellation factor

Sort-middle analysis

- Pros:
  - Redistribution occurs at a “natural” place
- Cons:
  - High communication cost if $T$ is high
  - Susceptible to load imbalance in the same way as sort-first
Sort-last

- Defers sorting until the end (imagine that)
- Renderers operate independently until the visibility stage
- Fragments transmitted over network to *compositing processors* to resolve visibility

Sort-last “flavors”

- **SL-sparse**
  - Only distributes pixels actually produced by rasterization
- **SL-full**
  - Stores and transfers a full image from each renderer
Sort-last analysis

- Overhead:
  - Boils down to the number of fragments that cross the wire times their size
    - SL-sparse overhead depends on number of pixels generated per frame
    - SL-full overhead depends on number of processors used (which is only indirectly related to number of primitives)

Sort-last analysis

- Pros:
  - Renderers implement full pipeline and are independent until pixel merging
  - Less prone to load imbalance
  - Very scalable (with SL-full at least)

- Cons:
  - Pixel traffic can be extremely high
So which is better?

- It depends. (Surprise, surprise.)
- Which ones can be best matched to hardware capabilities?
- Number of primitives, tesselation factor, coherence, etc., are all considerations. Many tradeoffs.

Future work

- Hybrid approaches seem promising
Designing Graphics Architectures Around Scalability and Communication

Eldridge, 2001

Much of the same

- Things I’m going to skim over:
  - Graphics pipeline overview
  - Types of parallelism
    - similar to Molnar’s description
    - note though that it calls out memory as a communications medium over time
  - Pomegranate architecture
    - to be presented (by me) later this term
What’s new

So what *do* we get out of this paper that interests us in today’s discussion?
- Further subdivision in our taxonomy
- Some pretty graphs to look at ;)

Revised Taxonomy: Overview

<table>
<thead>
<tr>
<th>Molnar</th>
<th>Revised</th>
<th>Pipeline-centric</th>
<th>Data-centric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort-first</td>
<td>Sort-first</td>
<td>Geometry</td>
<td>Vertex</td>
</tr>
<tr>
<td>Sort-middle</td>
<td>Sort-middle</td>
<td>Rasterization</td>
<td>Primitive</td>
</tr>
<tr>
<td>SL-sparse</td>
<td>SL-fragment</td>
<td>Fragment</td>
<td>Fragment</td>
</tr>
<tr>
<td>SL-full</td>
<td>SL-image</td>
<td>Display</td>
<td>Sample</td>
</tr>
</tbody>
</table>
Sort-last revisited

- Claims that “lumping together” SL-sparse and SL-full is unwarranted as they are not closely related
- Fragment sorting does not suffer the same disadvantages as image-composition sorting

Sort-first

- Sort-first retained-mode
  - Exploits frame-to-frame coherence
  - Challenging to implement
  - Sensitive to the amount of motion, amount of instancing, etc.
  - Example: Princeton’s Scalable Display Wall
Sort-first

- Sort-first immediate-mode
  - Can use unmodified commodity graphics accelerators as rendering pipes
  - All that changes is a software layer is inserted between the app and the many-in-place-of-one renderers
  - Example: WireGL (cough =-)

Sort-middle

- Sort-middle interleaved
  - Uses interleaved image parallelism
  - Each primitive is broadcast to all rasterizers
  - Example: SGI’s InfiniteReality
  - Problem: geometry performance not scalable
Sort-middle

- Sort-middle tiled
  - Uses tiled image parallelism
  - Primitives are routed rather than broadcast
  - Further subdivided:
    - Primitive Order (e.g., Stanford’s Argus, which uses one thread per screen tile)
    - Tile Order (e.g., Pixel-Planes 5, where tiles are processed sequentially)

Sort-last

- Sort-last fragment
  - Overlap factor guaranteed to be 1
  - But because geom. processing and rasterization are tied, there is no control over load balancing for rasterization
  - Examples: Evans & Sutherland Freedom Series, Kubota Denali
Sort-last

- Sort-last image composition
  - Forego ordered semantics
  - Load imbalance is extended to the fragment processors
  - However, fixed bandwidth is required (depends on size of output rather than input)

Hybrids

- WireGL + Lightning-2
- VC-1
Parallel Rendering
Crockett, 1995

What’s left to say?

- This paper provides mostly a conceptual overview
- Looks at other factors such as hardware details (shared vs. distributed memory, SIMD vs. MIMD, etc)
- Mentions binary swap
- Looks at non-rasterizing rendering methods (i.e., raytracing)