Parallel Polygon Rendering

Bengt-Olaf Schneider
NVIDIA

Outline

• Problem Definition and General Concepts
• Classification of Parallel Rendering Algorithms
• Load Balancing
• Practical Issues
  • Parallelizing Geometry Processing
  • Exploiting Graphics Hardware

The Problem

• Render a polygonal scene using multiple processors and/or threads
• Parallelization possible in every stage and overlapping execution of stages
• Each stage may be implemented on the same or on separate processors

Rendering Scenarios

• Rendering
  • Still-image rendering (single-frame)
  • Animation (multiple-frames)
  • Rendering algorithm and rendering quality
• Parallel Architectures
  • Tightly-coupled SMP
  • Loosely-coupled Clusters
  • Graphics accelerators or software renderers

MIMD, SIMD, Pipelining

• Multiple-Instruction, Multiple-Data (MIMD)
  • High level of independence between processors
  • Requires synchronization between processors
  • Best for tasks requiring many data-dependent branches
  • Used for geometry processing
• Single-Instruction, Multiple-Data (SIMD)
  • Tight coupling between processors
  • Implicit synchronization, good scalability
  • Best for tasks requiring same operations for many data
  • Used for rasterization and fragment processing
• Pipelining a.k.a. functional parallelization
  • Natural for many algorithms, e.g., graphics pipeline
  • Limited scalability and difficult to load balance

Classification of Parallel Renderers
Object vs. Image Partitioning

- **Object-space partitioning**
  - Most often used for geometry processing
  - Each task handles part of the scene, e.g. high-level objects, triangles etc.

- **Image-space partitioning**
  - Most often used for rasterization
  - Each task processes parts of the image, e.g. pixels, scanlines, tiles, etc.

Temporal Partitioning

- **Popular choice for rendering of animations**
- **Process**
  - Broadcast data to all processors
  - Assign frames (i.e. viewpoint etc.) to \( n \) processors
  - Render \( n \) frames
  - Display or store \( n \) frames
- **Considered to be embarrassingly parallel**
  - Rendering performance scales linearly with number of processors
  - But: Data distribution and collection (communication), forms serial overhead and may overshadow computation and rendering ...
  - no speedup & bad scalability !!

Classification by Sorting

- **Typical parallel polygon rendering system**
  - Geometry processing performed in object space, i.e. each processor handles a subset of the scene
  - Rasterization done in image space, i.e. each processor handles a subset of the screen
- **Objects must be sorted**
  - from object space into image space
  - Classification by where the sorting occurs.

Sort-Middle Rendering

- **Operation**
  - Round-robin assignment of objects to geometry proc's
  - Sort objects to screen regions
  - Rasterizers responsible for screen regions
- **Properties**
  - Good load-balancing during geometry processing (clipping!)
  - Possible load-balancing problems in rasterizer
  - Most popular scheme

Sort-Last Rendering

- **Operation**
  - GR-pair for full screen for all obj's
  - Compositing step determines visible objects / pixels
- **Properties**
  - Generally good load balancing
    (clipping, object size in screen space !)
  - No temporal order, transparency and anti-aliasing hard
  - E.g. PixelFlow

Sort-First Rendering

- **Quick mapping to screen regions and sorting**
- **Properties**
  - Less sorting for highly tesselated objects
  - Potential for load-imbalance due to uneven spatial distribution of objects
  - No actual system (yet).
**Load Balancing**

- Objective is to ...
  - distribute work evenly among all processors
  - have all processors finish their work at the same time

- (One) Definition of load balance:
  \[ \text{LB} = \frac{T_f}{T} \]
  - \( T_f \): Time fastest processor finishes
  - \( T \): Total processing time

**Load Balancing Problems**

- Geometry Processing
  - Clipping: Trivial accept/reject simpler than actual clipping
  - Vertex Count: Polygons have different number of vertices (e.g. clipping)
  - Tessellation: Higher order primitives may be tessellated into different number of basic primitives (triangles)
  - Rendering parameters: Number of lights, shading type, texturing algorithm etc.
  - Rasterization
    - Spatial distribution: Objects tend to be clustered in certain screen areas
    - Primitive size: Rasterization time is proportional to primitive size

**Load Balancing Issues**

- Terminology
  - Task: Basic unit of work that can be assigned to a processor, e.g. objects, primitives, scanlines, pixels, etc.
  - Granularity
    - Minimum number of tasks that are assigned to a processor, e.g. 10 scanlines or 128x128 pixel regions.
  - Coherence
    - Neighboring elements in space or time are similar, e.g. frames, scanlines, pixels
    - Exploited to speed up rendering calculations, e.g. for rasterization
    - Parallelization may destroy / hide coherence for a given processor

**Load Balancing Strategies**

- Static
  - Fixed assignment of tasks to processors

- Dynamic
  - On-the-fly assignment of tasks to processors

- Adaptive
  - Assign tasks such that all processors have approximately the same load

**Static Load Balancing**

- All tasks assigned before start of rendering, e.g.
  - Round-robin object assignment in sort-middle architectures
  - Assignment of screen regions to rasterizers (SGI R5100, Pixel planes 4)
  - Relies on assumptions about statistics of the model to achieve load balancing, e.g.
    - Most objects requires same amount of work to process
    - Interleaving of pixels will give each processor equal share of busy and less busy screen regions
    - All frames of an animation incur approximately same workload

**Dynamic Load Balancing**

- Task are assigned on demand, i.e. the next task goes to the first available processor
  - Assume that there are more tasks than processors
  - Granularity Ratio = \#tasks / \#processors > 1
  - Upper bound for load imbalance is difference between largest and smallest task
  - Simple optimization: (if known) assign largest tasks first
  - Task-processor assignment not known a priori
    - Maintains a task list that is depleted by processors
    - Requires dynamic (sic) distribution of tasks during runtime
    - Tasks may not complete in same order as issued
    - Some APIs require temporal ordering, e.g. OpenGL
### Rasterization modules for 128 x 128 pixel regions
- 80 regions for 1280 x 1024 display
- Several modules in a system

- Idle rasterizers process next unprocessed region
- Scalability for cost, performance and display size

### Adaptive Load Balancing

- Create tasks which will require (approximately) the same amount of processing time
- Static adaptive load balancing
  - Predictive: Estimate the processing time for each task
  - Reactive: Deduce processing time from previous frame
  - Requires separate step to determine task assignments

- Dynamic adaptive load balancing
  - Monitor workload of processors
  - Reassign tasks from busy processors to idle processors
  - Requires concurrent monitoring process

- Let’s look at concrete load balancing algorithms:

### Adaptive Load Balancing Algorithms

- Usually performed in 2 steps:
  - Load estimation
    - Count primitives per screen region
    - Estimate cost per primitive
  - Work distribution
    - Subdivide (tile) screen to create regions with approximately equal load, or...
    - Assign fixed-sized regions (cells) to processors to create approximate load balance

### Roble’s Method

- Determine number of primitives per region
- Combine low-load regions
- Split high-load regions in half
- No control over location of split and hence chance for low/no effectiveness

### Whelan’s Method (1992)

- Tally primitives overlapping each region
- Build quadtree of all regions, assigning number overlapping primitives at each quadtree node
- Top-down subdivision of quadtree for nodes with highest primitive count
- May still leave unbalanced work distribution
  - Increased granularity ratio
  - Big primitives are counted multiple times
  - Overestimation of work

### Whelan’s Method

- Count number of centroids per region
- Split regions using distribution of centroids
  - Median-cut algorithm to subdivide such that both new regions contain same number of centroids
- Large effort for sorting primitives
- No accounting for primitive size

- Initial subdivision into tiles for given granularity ratio (here 2)
- During rendering, processors who finish early "steal" work from busy processors by splitting work region
  - An idle processor finds the processor with most work left
  - Split only if remaining work exceeds a threshold

Mueller's Method: MAHD

- Mesh-based adaptive hierarchical decomposition
  - Based on small screen cells, i.e. a fine, regular mesh
  - Count primitives overlapping each cell in inverse proportion to their size
  - Avoid accounting problems like in Whitman's method
  - Heuristic to balance constant geometry cost and size-dependent rasterization cost (experimentally justified)
  - Build a summed-area table
  - Subdivide screen into regions along cell boundaries
  - Similar to median-cut algorithm but cheaper because of SAT

Ellsworth's Method

- Reactive method, load balancing between frames
- Tally number of primitives overlapping each cell
- Estimate processing time for each cell
- Greedy, multiple-bin-packing algorithm to assign regions to processors
  - Sort regions by descending polygon counts
  - Assign next region to processor with lightest workload

Princeton Display Wall

- Multi-projector system
  - Each projector is driven by a dedicated computer
  - Computers are network-connected
  - Load-balancing is achieved by dividing the projection screen into virtual, non-overlapping tiles
  - Sort-first architecture
  - Each computer stores & renders all objects in a virtual tile
  - Right before display, pixel data are sent to the computer who owns the corresponding screen tile
  - Coarse-grain workload decomposition
  - Requires fast frame-buffer reads and fast networks
  - Assignment strategy must balance tile-object overlaps (few large tiles) and load-balancing (many small tiles)

Princeton Display Wall (2)

- Grid Bucket Assignment
  - Greedy algorithm using grid cells smaller than screen tile
  - For each tile compute cost for rendering & transmitting a tile
  - Distribute tiles from the server with the largest workload to the server with the smallest workload
- KD-Split
  - Subdivide the screen into exactly P regions for P computers to minimize overhead
  - Chose partition to balance workload between two halves
  - Subdivide each half further
  - Better workload estimate overcomes some of the problems with similar approaches (Roble, Whelan, Whitman 1992)

IBM Yotta

- Adaptive sort-first architecture
  - Multiple graphics adapters in separate computers (SP nodes) drive single display
  - Screen divided in bands, assigned to graphics adapters
  - Reactive adjustment of bands according to work load
- Graphics outputs composited for final display
  - Outputs synchronized for stereo support
Parallel Geometry Processing

- Geometry Processing
  - Transformations, Lighting, Clipping, Texture Calculations
- Round-robin distribution
  - Each processor works on $\frac{1}{n}$ objects
  - Objects are sent to rasterizers as soon as possible
  - Load balancing problems as objects may require different amounts of work (clipping, lighting, ...)
- Dynamic assignment
  - First available processor receives next object
- Limitations of these strategies:
  - Temporal object ordering is lost

- Required by some APIs, e.g. OpenGL
- Important for algorithms that rely on the order in which objects are drawn onto the screen
  - Non-z-buffer hidden surface removal, e.g. painter's algorithm
  - Multi-pass algorithms, e.g. transparency, overlays, solid-modeling, priority algorithm
- Possible solutions
  - Sequence numbers (time stamps) enforce strict ordering
  - Barriers to ensure ordering between groups of objects (see Igley, Stoll and Hannahan, Siggraph 98)

Temporal Ordering

- Parallel Polygon Rendering Hardware
  - Typically sort-middle architecture
  - Rasterization and setup always in hardware
  - Geometry calculations either in hardware or in software
- Parallel Hardware Rasterization
  - Most systems interleave pixels or scanlines, e.g. SGI RE/IR, 3Dlabs, Integraph
  - Some systems overlay a coarser tiling scheme for virtual rasterizers, e.g. Pixel-planes 5, Talisman, PowerVR, Oak

Exploiting Graphics Hardware

- Graphics accelerators are very useful for single-processor rendering
- Multiple CPUs and/or multiple graphics adapters are a challenge
  - SMP geometry pipelines are difficult for many APIs
  - In clusters, communication often overshadows computation and rendering
  - Communication overhead is often overlooked or excluded
  - Even “embarrassingly parallel” approaches (frame-parallel) are not trivially implemented
  - Require high-speed networks and high-speed disks
  - Poor scaling and chance for deceleration
  - Solution: Increase CPU load to reduce communication

- Hardware can shorten rendering time
  - Shift balance between computation & communication
  - Rebalance by reducing comm. overhead
  - Use more CPU, e.g. compression, selective updates, ...
  - Use less bandwidth, e.g. distributed frame buffers
  - Overlap communication and computation

<table>
<thead>
<tr>
<th></th>
<th>G</th>
<th>R</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>No HW</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With HW</td>
<td>G</td>
<td>R</td>
<td>C</td>
</tr>
<tr>
<td>Rebalanced</td>
<td>G</td>
<td>R</td>
<td>C</td>
</tr>
</tbody>
</table>
Avoid communication of final pixel values to a central storage or display.
Instead use dedicated merge hardware to combine partial image.

- e.g. Wei et al., PRS95 or Yotta.

Distributed Frame Buffer

From Parallel Renderers

FB 1  FB 2  FB 3  FB 4

To Display

Merge

Summary

- Parallel polygon rendering classifications
  - Sort-first/middle/last

- Key problems in parallel polygon rendering
  - Load balancing
  - Communication overhead

- Load Balancing
  - Required due to uneven work distribution
  - Static, dynamic, adaptive

- Communication overhead
  - Distribution and collection of data
  - Minimize transfers or support in hardware