Lecture 3:
Sort-Everywhere Pipeline Scheduling and Geometry Processing

Visual Computing Systems
CMU 15-869, Fall 2013
Scheduling a sort-everywhere graphics pipeline

The following figures follows the design of Pomegranate [Eldridge et al.], but use modern graphics pipeline stage names)
Starting state: draw commands enqueued for pipeline

Input: three triangles to draw
   (fragments to be generated for each triangle are shown below)

Draw T3
Draw T2
Draw T1

Frame buffer

CMU 15-869, Fall 2013
After geometry processing, then assign first two processed triangles to rast 0.

Input:

- Draw T1
- Draw T2
- Draw T3

Frame buffer
Assign next triangle to rast 1 (round robin policy, batch size = 2)

Q. What is the ‘next’ token for?

Input:

Draw T1 → 1 2 3 4
Draw T2 → 1 2 3 4
Draw T3 → 1 2 3

Frame buffer
**Rast 0 and rast 1 can process T1 and T3 simultaneously**

(Shaded fragments enqueued in frame-buffer unit input queues)

Input:

---

Draw \( \Delta T_1 \) \( \rightarrow \) 1 2 3 4
Draw \( \Delta T_2 \) \( \rightarrow \) 1 2 3 4
Draw \( \Delta T_3 \) \( \rightarrow \) 1 2 3

Frame buffer

---

CMU 15-869, Fall 2013
FB 0 and FB 1 can simultaneously process fragments from rast 0
(Notice updates to frame buffer)
Fragments from T3 cannot be processed yet. Why?

Input:

Draw $\triangle^T_1$ → 1 2 3 4

Draw $\triangle^T_2$ → 1 2 3 4

Draw $\triangle^T_3$ → 1 2 3

Frame buffer
Rast 0 processes T2
(Shaded fragments enqueued in frame-buffer unit input queues)

Input:

<table>
<thead>
<tr>
<th>Draw</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Frame buffer

0 1
1 0
T1,1
T1,2
T1,3
T1,4
Rast 0 broadcasts ‘next’ token to all frame-buffer units

Input:

Draw T1,1 → 1 2 3 4
Draw T1,2 → 1 2 3 4
Draw T1,3 → 1 2 3

Frame buffer
FB 0 and FB 1 can simultaneously process fragments from rast 0
(Notice updates to frame buffer)
Switch token reached: frame-buffer units start processing input from rast 1
FB 0 and FB 1 can simultaneously process fragments from rast 1
(Notice updates to frame buffer)
Extending to parallel geometry units
Starting state: commands enqueued

Input:

Draw T1: 1 2 3 4
Draw T2: 5 6 7
Draw T3: 1 2 3 4
Draw T4: 1 2

Frame buffer:
Distribute triangles to geom units round-robin (batches of 2)

Input:

Draw \( T_1 \) → \[ \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 \end{array} \]

Draw \( T_2 \) → \[ \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 \end{array} \]

Draw \( T_3 \) → \[ \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 \end{array} \]

Draw \( T_4 \) → \[ \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 \end{array} \]
Geom 0 and geom 1 process triangles in parallel
(Result after T1 processed are shown. Note big triangle T1 broken into multiple work items. [Eldridge et al.])
Geom 0 and geom 1 process triangles in parallel
(Triangles enqueued in rast input queues. Note big triangles broken into multiple work items. [Eldridge et al.])
Geom 0 broadcasts token to rasterizers

Geometric 0

Geometry 1

Rasterizer 0

Rasterizer 1

Frag Processing 0

Frag Processing 1

Frame-buffer 0

Frame-buffer 1

Input:

Draw $\overrightarrow{T_1}$ → 1 2 3 4

Draw $\overrightarrow{T_2}$ → 1 2 3 4

Draw $\overrightarrow{T_3}$ → 1 2 3 4

Draw $\overrightarrow{T_4}$ → 1 2

0 1
1 0

Frame buffer
Rast 0 and rast 1 process triangles from geom 0 in parallel
(Shaded fragments enqueued in frame-buffer unit input queues)

Input:

<table>
<thead>
<tr>
<th>Draw</th>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>T2</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>T3</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>T4</td>
<td>1 2</td>
</tr>
</tbody>
</table>

Frame buffer:

```
0 1
1 0
```
Rast 0 broadcasts token to FB units

Distrib

Geometry 0
  | Switch
  | T1,5, T1,3, T1,1
  | Next
  | T3,b, T3,a

Geometry 1
  | Switch
  | T4

Rasterizer 0

Frag Processing 0
  | Switch
  | T1,5, T1,3, T1,1
  | T2,3, T2,1, T1,6

Frag Processing 1
  | Switch
  | T1,4, T1,2
  | T2,4, T2,2, T1,7

Frame-buffer 0

Frame-buffer 1

Input:

Draw T1
  → 1 2 3 4

Draw T2
  → 1 2 3 4

Draw T3
  → 1 2 3 4

Draw T4
  → 1 2

Frame buffer
Frame-buffer units process frags from rast 0 in parallel
(Notice updates to frame buffer)

Input:

Draw $\rightarrow$ 1 2 3 4

Draw $\rightarrow$ 5 6 7

Draw $\rightarrow$ 1 2 3 4

Draw $\rightarrow$ 5

Draw $\rightarrow$ 1 2

Frame buffer
Switch token reached by FB: FB units start processing input from rast 1
Switch token reached by rast units: rast units start processing input from geom 1

Switched tokens:
- T1
- T2
- T3
- T4

Input:

Draw T1 → 1 2 3 4
Draw T2 → 1 2 3 4
Draw T3 → 1 2 3 4
Draw T4 → 1 2

Frame buffer:

0 1
1 0

T1,1
T1,2 T1,3
T1,4 T1,5
Rast 0 processes triangles from geom 1
(Note Rast 1 has work to do, but cannot make progress because its output queues are full)
Rast 0 broadcasts ‘next’ token to frame-buffer units

Input:

Draw $\Rightarrow \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & \\ \end{array}$

Draw $\Rightarrow \begin{array}{cccc} 1 & 2 & 3 & 4 \\ \end{array}$

Draw $\Rightarrow \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & \end{array}$

Draw $\Rightarrow \begin{array}{cccc} 1 & 2 \end{array}$
Frame-buffer units process frags from rast 1 in parallel
(Notice updates to frame buffer. Also notice rast 1 can now make progress since space has become available)
Switch token reached by rast: rast units start processing input from rast 0

Input:

Draw → 1 2 3 4
Draw → 5 6 7 4
Draw → 1 2 3 4
Draw → 5
Draw → 1 2

Frame buffer

0 1
1 0
T1,1 T2,1 T2,2
T1,2 T1,3 T2,3 T2,4
T1,4 T1,5 T1,6 T1,7
Frame-buffer units process frags from rast 0 in parallel
(Notice updates to frame buffer)

Input:

- Draw \( \text{T1} \) → 1 2 3 4
- Draw \( \text{T2} \) → 1 2 3 4
- Draw \( \text{T3} \) → 1 2 3 4
- Draw \( \text{T4} \) → 1 2

Frame buffer:

```
0 1 T3,1 T3,2
1 0 T3,3 T3,4
T1,1 T3,5 T2,1
T1,2 T1,3
T1,4 T1,5 T1,6
T1,7
```
Switch token reached by rast: rast units start processing input from rast 1
Frame-buffer units process frags from rast 1 in parallel
(Notice updates to frame buffer)

Input:

Draw \( \rightarrow \) 1 2 3 4

Draw \( \rightarrow \) 1 2 3 4

Draw \( \rightarrow \) 1 2 3 4

Draw \( \rightarrow \) 1 2

Frame buffer
Geometry Processing Basics
(Focus on design and implementation challenges of tessellation)
Review: coordinate spaces

- Object’s modeling transform: compute surface vertex coordinate in world-space (object-to-world)
- View transform: compute vertex coordinate in camera-space (world-to-camera)
- Projection transform: compute vertex coordinate in homogeneous clip space (camera-to-clip)

Camera-space 3D view frustum:
- Camera at origin
- Camera looking down -Z axis
- All triangles intersecting with this region will be drawn. All triangles outside the region will be discarded prior to rasterization (they off-screen)

Canonical clip-space view frustum:
- (-1,-1,-1) to (1,1,1) in euclidian 3D space
- (-w,-w,-w, *) to (w,w,w, *) in homogeneous 3D space

Clipping example: triangle extends beyond view volume. Clipped to form two triangles entirely within volume.
Per vertex operations: application-specified vertex shader computes vertex positions

Vertex positions emitted by vertex processing (or the geometry shader, if enabled) are represented in homogeneous, clip-space coordinates.

Homogeneous vertex \( v = (x, y, z, w) \)

Vertex position in euclidian space: \( (x/w, y/w, z/w) \)

Vertex is within the canonical clip-space view frustum if:

\[-w \leq x \leq w\]
\[-w \leq y \leq w\]
\[-w \leq z \leq w\]
Per-primitive operations

Given input vertices (with positions in homogeneous clip space):

1. “Assemble” vertices into primitives
2. Discard primitive if outside of view frustum
3. Clip primitive against view frustum (if clipping is necessary)
4. For each resulting primitive
   - Divide homogeneous x,y,z coordinates by w (coordinates now in [-1,1] range)
   - Apply viewport transform to compute screen position of vertex (coordinates now in [0,screen_width] range)
   - Discard back-facing primitives [optional, depends on config]
Assembling vertices into primitives

How to assemble vertices into primitives is pipeline state (specified by draw command)

Notice: independently processed vertices get grouped into primitives ("join"!)

GL_TRIANGLES

GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN
Clipping

- May generate new vertices/primitives, or eliminate vertices/primitives
- Data-dependent computation
  - Variable amount of work per primitive
  - Variable control flow per primitive
Why clipping?

- Avoid downstream processing that will not contribute to image (rasterization, fragment processing)

- Establish invariants for emitted primitives
  - Can safely divide vertex clip space xyz coordinates by w after clipping
  - Bounds vertex positions (can now choose precision of subsequent operations accordingly)
Guard-band clipping

- Idea: clips against wider X-Y plane clip bounds than canonical view volume
- Reduces number of triangles that must undergo clipping -- reduces variance in per-primitive clipping work
- Cost is that primitives no longer guaranteed to be fully on screen
  - Rasterizer must not generate off-screen fragments
  - Increased precision required during rasterization

[RealityEngine, Akeley 93]
Back-face culling

- Use sign of triangle area to determine if triangle is facing toward or away from camera
- May discard primitive as a result of this test
  - For closed meshes, eliminates ~ 1/2 of triangles
    (these triangles would be occluded by other triangles anyway)

Triangle area = \frac{(x_0y_1 - x_1y_0) + (x_1y_2 - x_2y_1) + (x_2y_0 - x_0y_2)}{2}
Tessellation Pipeline
Graphics pipeline support for tessellation

- No tessellation
  - Regular character (2K triangles rendered): 874 fps

- Tessellated surfaces with displacement mapping
  - Tessellated character (1 Million triangles rendered): 120 fps
  - ~500x increase in detail for 7x cost in speed

Image credit: 3D Guru (from NVIDIA)

[Tatarchuk 2007]
No tessellation

Low detail

Image credit: Unigine Engine, copyright Unigine Corp.
Tessellated surfaces with displacement mapping

Image credit: Unigine Engine, copyright Unigine Corp.
Modern GPU tessellation

**Motivations:**
- Reduce footprint (do not store high resolution triangles in memory)
- For animated surfaces: reduce CPU-GPU bandwidth (do not transfer high resolution mesh over PCIe bus to GPU each frame)
- Animate/skin/compute physics using course resolution mesh, but render high resolution mesh

**Design:**
- Hybrid programmable, non-programmable implementation (three new pipeline stages)
- Supports only parametric surfaces (surface must support direct evaluation of \( f(u, v) \))
Parametric surface

3D surface position is a function of 2D parametric coordinate $(u,v)$

$[0,1]^2$ parametric domain

$$f(u,v) = (x,y,z)$$
Example parametric surfaces

Bicubic patch, 16 control points
(quad domain)

See “Approximating Catmull-Clark
Subdivision Surfaces With Bicubic
Patches”, Loop et al. 2008

PN Triangles, 3 vertices + 3 normals
(defines bezier patch on triangular domain)

See “Curved PN Triangles”,
Vlachos et al. 2008
Tessellation pipeline

- **Hull shader**
  - Programmable stage
  - Input: assembled primitives after traditional vertex processing
  - Output: new “patch” primitive
    - Tessellation factor for each edge of parametric domain
    - Control points for parametric surface (computed from input primitive vertices)

Note: D3D11 Stage Naming (not canonical stage names)
Hull shader produces edge tessellation rates

Based on estimate of parametric surface position

(Note: rates need not be integral)
Fixed-function tessellation stage

Input: edge tessellation constraints for a patch
Output: (almost) uniform mesh topology meeting constraints

[Moreton 01]
Domain shader stage

Input: control points (from hull shader) and stream of parametric vertex locations \((u,v)\) from tessellator

Output: position of vertex at parametric coordinate: \(f(u,v)\)
Modern GPU tessellation

- **Hull shader (programmable)**
  - Coarse primitive granularity
  - Data-parallel across coarse primitives
  - Large working set per invocation (typically a primitive + one-ring)

- **Tessellator**
  - Surface agnostic, fixed-function hardware implementation
  - Irregular control flow

- **Domain shader (programmable)**
  - Fine-mesh-vertex granularity
  - Data-parallel (preserves shader programming model)
  - Direct evaluation of surface at specified domain coordinate
Challenge: avoid cracks!

Hull shader must carefully compute edge tess factors to avoid cracks.
Modern GPU tessellation summary

- Enables adaptive level-of-detail, high-resolution meshes in games

- Heterogeneous, 3-stage abstraction: co-design of algorithm and hardware
  - Retain flexibility: surface type is programmable, but must be parametric
  - Minimize new complexity: extension of existing shader programming model
  - Retain performance:
    - Data-parallel evaluation of fine vertex positions
    - Fixed-function HW algorithm for generating points (not data parallel)

- Challenges
  - Application developer: avoiding cracks (requires consistent edge rate evaluation -- this is tricky in floating point math)
  - GPU implementor: managing large data amplification... while maintaining parallelism, locality, and order
Programmable geometry amplification

- Clipping amplifies geometry (clipping can result in multiple output primitives)

- Amplification by geometry shader stage or tessellation stages in a modern pipeline is far greater than that of clipping

- Geometry shader: output up to 1024 floats worth of vertices per input primitive

- Tessellation: thousands of vertices from a single base primitive (challenge to maintain highly parallel execution)
Thought experiment

Round robin distribution of primitives to geometry pipelines
Thought experiment
Approach 1: make on-chip buffers as big as possible, but tolerate stalls
- Run fast for low amplification (never move queue data off-chip)
- Run very slow under high amplification (serialization of processing due to blocked units)
Thought experiment

Approach 2: never block geometry unit: allocate worse-case space off chip
- Run slower for low amplification (data goes off chip then read back in by rasterizers)
- No performance cliff for high amplification (still maximum parallelism)
Hybrid approach: allocate “average case” space on chip, spill to memory only if necessary
- Run fast for low amplification (high parallelism, no memory traffic)
- Less of performance cliff for high amplification (high parallelism, but incur memory traffic)
NVIDIA GPU implementation

Optionally resort work after Hull shader (since amplification factor known)