Evolving the Direct3D Pipeline for Real-time Micropolygon Rendering

Kayvon Fatahalian
Stanford University
A real-time micropolygon rendering pipeline ... is not far away.

High-throughput micropolygon rasterization
[Fatahalian et al. 2009]
[Brunhaver et al. 2010]

DiagSplit: parallel, adaptive tessellation
[Fisher et al. 2009]

Micropolygon occlusion culling
[Boulos et al. 2010]

Quad-fragment merging
[Fatahalian et al. 2010]
It is inefficient to render micropolygons using the Direct3D graphics pipeline implemented by GPUs.
Micropolygons pose problems for real-time pipeline

**TESSELLATION**
Direct3D is missing high-quality, adaptive tessellation

**RASTERIZATION**
Big-triangle optimizations (like pixel-parallel coverage tests) are not efficient for micropolygons

**SHADING**
Pipeline generates over eight times more shading work than needed

How does the real-time graphics pipeline evolve to enable efficient micropolygon rendering?
TESSELLATION
Tessellation output: micropolygon mesh

Goal: all triangles are approximately 1/2 pixel in area
(yields about one vertex per pixel: sample detailed surface accurately)
Uniform patch tessellation is insufficient

Uniform partitioning of patch (parametric domain)

Patch viewed from camera

Too many polygons: poor performance

Polygons too large: poor quality
Adaptive tessellation:

Lane-Carpenter patch algorithm ("split-dice")

[Lane 80]
Adaptive tessellation (divide-and-conquer)

Patch parametric domain

Patch viewed from camera
Adaptive tessellation (divide-and-conquer)

Patch parametric domain

Patch viewed from camera
Adaptive tessellation (divide-and-conquer)

Patch parametric domain

Patch viewed from camera
Unaligned tessellations cause cracks

(parametric domain)
Crack-free, (almost) uniform tessellation in Direct3D

Input: edge tessellation constraints for a patch
Output: (almost) uniform mesh that meets these constraints

[Moreton 01]
Direct3D 11 tessellation pipeline

No adaptive partitioning of patches

Uniform tessellation (mesh generation)

base patch data + edge rates

tessellated meshes

Vertex Processing

final vertex positions
Adaptive tessellation pipeline

1. **Split**
   - base patch data
   - sub-patches + edge rates

2. **Uniform tessellation (mesh generation)**
   - sub-patch meshes

3. **Vertex Processing**
   - final vertex positions

**Options: Fixed-function, Programmable**
New algorithm: DiagSplit

Use non-isoparametric “splits” to generate sub-patches with matching edge tessellations

DiagSplit: Fisher et al. SIGGRAPH Asia 2009

(parallel, crack-free, adaptive tessellation)
DiagSplit tessellation pipeline

Base patch data

DiagSplit

Compute edge rates

Surface eval(u,v)

sub-patches + edge rates

Uniform tessellation (mesh generation)

sub-patch meshes

Vertex Processing

final vertex positions

Fixed-function

Programmable
Lessons learned #1:
The divide-and-conquer component of DiagSplit is very useful.
Benefit #1: DiagSplit reduces vertex count

- Up to eight times fewer vertices for large, foreshortened patches
- Vertex processing is 70-80% of tessellation cost (splitting pays for itself)
Benefit #2: splitting implicitly creates a hierarchy over surface triangles

- Provides opportunity to cull surfaces at varying scales
  - It is beneficial to occlusion-cull sub-patches after each split
    [Boulos et al. HPG 2010]

- Perform splits (a.k.a. traverse hierarchy) in depth-first order
  - Small working set compared to breath-first tessellation: stream the geometry
  - Emits triangles in an order with high-spatial locality
RASTERIZATION
Rasterization

Compute coverage using point-in-triangle tests
Key optimizations

1. Data-parallel point-in-triangle tests

2. Trivial “accept”
Micropolygons: most point-in-triangle tests fail

61% of candidate samples inside triangle

6% of candidate samples inside triangle

Bounding box of micropolygon is tight

Low sample test efficiency!
## Micropolygon Rasterization

For each MP

<table>
<thead>
<tr>
<th>Setup</th>
<th>Cull polygon if back-facing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bound</td>
<td>Compute subpixel bbox of MP</td>
</tr>
<tr>
<td>Test</td>
<td>For each sample in bbox</td>
</tr>
<tr>
<td></td>
<td>Test MP-sample coverage</td>
</tr>
</tbody>
</table>

Thursday, July 29, 2010
Parallel micropolygon rasterization

Process many micropolygons simultaneously

Input micropolygons

Output fragments
Micropolygon rasterization is simple, but expensive

- 28% of tested samples fall within the triangle
  - Good: Up from 11% from a 16-sample-stamp algorithm
  - Bad: Still much lower than stamp-based algorithms on large triangles

- No cheap “all-in” cases

- Can’t amortize setup across many sample tests
1 billion micropolygons/sec at 16x MSAA
(~15 million polygon scene at 60 Hz)

Estimated cost of GPU software implementation in CUDA:

About seven high-end NVIDIA GPUs

Note: panel discussion on utility of fixed-function hardware later today!

See [Brunhaver et al. HPG 2010]: A Hardware Implementation of Micropolygon Rasterization...
Stochastic micropolygon rasterization (motion blur)

- Increases rasterization costs by 3-7x
  - More point-in-triangle tests (only 5% of tested samples lie within polygon)
  - Individual tests are more expensive

- Also need smarter occlusion culling for moving objects
  [Space-time hierarchical occlusion culling: Boulos et al. 2010]
Lessons learned #2:
Real-time micropolygon rendering begs for fixed-function rasterization.
(it is simple, but expensive)
Lessons learned #3:

Algorithms for rasterizing micropolygons with motion blur are surprisingly inefficient (only 1 in 20 point-in-triangle tests contribute to image).

(interesting time to evaluate alternatives, like ray tracing)

But evaluate them in the context of FUTURE rendering conditions!
Like adaptive tessellation and motion blur!
SHADING
Current GPUs shade small triangles inefficiently
Pixels at triangle boundaries are shaded multiple times

Shading computations per pixel

- 8+
- 7
- 6
- 5
- 4
- 3
- 2
- 1
Pixels at triangle boundaries are shaded multiple times

Shading computations per pixel
Pixels at triangle boundaries are shaded multiple times

Shading computations per pixel

Thursday, July 29, 2010
Small triangles result in extra shading

100 pixel area triangles

10 pixel area triangles

1 pixel area triangles
Goal:
Shade high-resolution meshes (not individual triangles) approximately once per pixel

Solution:
Quad-fragment merging
GPU pipeline: triangle connectivity is known

Triangle connectivity is known

quad fragments
Pipeline with quad-fragment merging

Tess → Rast → Merge → Shade

[Fatahalian et al. SIGGRAPH 2010]
Pipeline with quad-fragment merging

Tess \rightarrow \text{merge buffer (32 quad-fragments)} \rightarrow \text{Rast} \rightarrow \text{Merge} \rightarrow \text{Shade}

Adjacent Triangles: 2
Adjacent Triangles: 1, 3
Adjacent Triangles: 2, 4
Adjacent Triangles: 3

[Fatahalian et al. SIGGRAPH 2010]
How to merge quad-fragments

Mesh triangles

Rasterized quad fragments

Merged quad fragment
When to merge quad-fragments

Challenge: avoiding merges that introduce visual artifacts
Example: surface with a silhouette

Triangle mesh

Final pixels

anti-aliased silhouette

Thursday, July 29, 2010
Naive merging results in aliasing

Only merge quad-fragments from adjacent triangles in mesh
Merging reduces total shaded quad fragments

1/2-pixel-area triangles: 8x reduction

Big Guy Scene

Shading computations / pixel (avg)

Average triangle area (pixels)

No merging

Merging
Merging reduces total shaded quad fragments

Ten-pixel-area triangles: 2x reduction

Big Guy Scene

Shading computations / pixel (avg)

No merging

Merging

Average triangle area (pixels)
Nearly identical visual quality *

Quad-fragment merging

Current GPU (no merging)

* see SIGGRAPH 2010 paper and talk slides for more detail on possible artifacts
Nearly identical visual quality

Quad-fragment merging

Current GPU (no merging)
Doesn’t deferred shading achieve one shade per pixel?

- Large storage and bandwidth costs
- Interacts poorly with multi-sample anti-aliasing and shader derivatives
  (no, Direct3D 10.1 multi-sample access doesn’t solve this)
What about Reyes-style shading?
(Sample shading at micropolygon mesh vertices prior to rasterization)

In comparison to Reyes, quad-fragment merging:

- Is an evolution of the GPU status quo
- Supports high-quality shading for all triangle sizes
- Leverages fine-granularity occlusion-culling mechanisms in GPUs
  (sometimes 2x better than culling at Reyes-grid granularity)

- Does not yet support motion blur (future work)
- Can exacerbate some screen-space shading artifacts (see paper)
Lessons learned #4:
(actually, it’s more of an interesting question)

Under what situations is it preferable to sample shading in screen space (like a GPU)? When is it better to sample in object space (like Reyes)?

Also see:
[Ragan-Kelley et al.] Decoupled Sampling: Extends GPU fragment-shading to motion blur
[Burns et al. HPG 2010] Extends Reyes to larger triangles, moves object-space shading after rasterization
SUMMARY
A real-time micropolygon rendering pipeline

**DiagSplit adaptive tessellation:**

- Reduces rendered vertex count
- Enables hierarchical occlusion culling
- Simplifies micropolygon-parallel rasterization
- Makes quad-fragment merging practical
  (provides topology, sets triangle order)

**Diagram:***

1. DiagSplit
2. Uniform Tess
3. Vertex Processing
4. Rasterization
5. Fragment Merging
6. Fragment Processing
7. Pixel Operations

*New!*
A real-time micropolygon rendering pipeline

Rasterization:

Simple, but expensive: fixed-function hardware highly desirable

Motion blur via stochastic sampling is an area of very active research (I believe this is a feature that will go into future GPUs)
A real-time micropolygon rendering pipeline

Quad-fragment merging:

For micropolygons: reduces shaded fragments by 8x
(2x benefit for ten-pixel-area triangles)

Output quality usually very close to that of current GPU

Future work seeks to extend technique to motion blur
A real-time micropolygon rendering pipeline ... is not far away.

DiagSplit: parallel, adaptive tessellation
[Fisher et al. 2009]

Micropolygon occlusion culling
[Boulos et al. 2010]

High-throughput micropolygon rasterization
[Fatahalian et al. 2009]
[Brunhaver et al. 2010]

Quad-fragment merging
[Fatahalian et al. 2010]
Suggested reading

Real-time Reyes-style Adaptive Surface Subdivision. Patney et al. SIGGRAPH Asia 2008

Data-Parallel Rasterization of Micropolygons with Defocus and Motion Blur. Fatahalian et al. HPG 2009
Hardware Implementation of Rasterization with Motion and Defocus Blur. Brunhaver et al. HPG 2010
Micropolygon Ray-tracing with Motion and Defocus Blur. Hou et al. SIGGRAPH 2010

Reducing Shading on GPUs using Quad-Fragment Merging. Fatahalian et al. SIGGRAPH 2010
Decoupled Sampling for Real-Time Graphics Pipelines. Ragan-Kelley et al. (in submission, see his website)
A Lazy Object-Space Shading Architecture With Decoupled Sampling. Burns et al. HPG 2010

Space-Time Hierarchical Occlusion Culling for Micropolygon Rendering With Motion Blur. Boulos et al. HPG 2010
Efficient Bounding of Displaced Bezier Surfaces. Munkberg et al. HPG 2010
Thank you

Micropolygon Project Contributors:

Kayvon Fatahalian
Solomon Boulos
John Brunhaver
Matthew Fisher
Edward Luong
James Hegarty

Kurt Akeley
William R. Mark
Henry Moreton
Pat Hanrahan