Evolving the Real-time Graphics Pipeline for Micropolygon Rendering

Kayvon Fatahalian
CS448s
May 25, 2010
Highly detailed surfaces

Credit: Pixar Animation Studios, UP (2009)
Highly detailed surfaces

Credit: Pixar Animation Studios, UP (2009)
Interactive graphics: low geometric detail
Interactive graphics uses large triangles

High-quality surfaces: all triangles in this bin!

[source NVIDIA]
Micropolygons

(one pixel)
It is inefficient to render micropolygons using the OpenGL/Direct3D graphics pipeline implemented by GPUs.
Sources of inefficiency

- Tessellation
  (micropolygon generation)
- Rasterization
- Shading
Missing: adaptive tessellation

Generate triangles on-demand in the pipeline

Input base patches

Micropolygon mesh
Rasterization: computing covered pixels
Micropolygons too small for pixel-parallelism
Shading: computing surface color
Micropolygons pose three big problems

**TESSELLATION**
Cannot adaptively tessellate a surface into micropolygons in parallel.

**RASTERIZATION**
Pixel-parallel coverage tests are inefficient.

**SHADING**
Pipeline generates over $8 \times$ more shading work than needed.
Two widely-used rendering pipelines

Pixar REYES (Cook 87)
- Primitive Split
- Dice
- Microvertex Shading
- Rasterization
- Pixel Operations

OpenGL (Akeley 92) / Direct3D (Blythe 05)
- Primitive Tessellation
- Vertex Processing
- Rasterization
- Fragment Shading
- Pixel Operations

+ many people exploring real-time RAY TRACING [recall: Optix]
How does the real-time graphics pipeline evolve to enable efficient micropolygon rendering?

**TESSELLATION**
DiagSplit: generating micropolygons with adaptive tessellation

**RASTERIZATION**
Building a micropolygon rasterizer and adding support for motion blur

**SHADING**
Why shading on a GPU is inefficient, and a way to fix it.
TESSELLATION:
Integrating parallel, adaptive tessellation into the pipeline

Fisher, Fatahalian, Boulos, Akeley, Mark, Hanrahan (SIGGRAPH Asia 2009)
Overview: current solutions

- **Lane-Carpenter patch algorithm**
  - High-quality, adapts well to surface complexity
  - Hard to parallelize

- **GPU tessellation**
  - Low quality, does not adapt well
  - High performance (parallel, fixed-function)
Tessellation input: parametric patches

Input base patches
(example: bicubic patch)

[Vlachos 01, Loop 08, Loop 09]
Tessellation output: micropolygon mesh

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

Uniform partitioning of patch (parametric domain)

Too many polygons: poor performance

Polygons too large: poor quality

Patch viewed from camera
Adaptive tessellation:

Lane-Carpenter patch algorithm

[Lane 80]
Adaptive tessellation

Patch parametric domain

Patch viewed from camera
Adaptive tessellation

Patch parametric domain

Patch viewed from camera
Adaptive tessellation

Patch parametric domain

Patch viewed from camera
Off-line status quo: “stitching” fixes cracks

Use a strip of polygons to connect adjacent sub-patches

Creates dependency: cannot process sub-patches in parallel
Parallel crack fixing

Adjacent regions agree on tessellation along edge
(in this case: 5 segments)

T(edge) = 5
Crack-free, uniform tessellation

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

Crack-free, uniform patch tessellation
But no adaptive partitioning of patches!

Base patch data + edge constraints

Uniform tessellation (mesh generation)

Mesh topology + parametric location of vertices

Vertex Processing

final vertex positions
Want: adaptive tessellation pipeline

Base patch data

Adaptive partitioning

Sub-patch data + edge constraints

Uniform tessellation (mesh generation)

Mesh topology + parametric location of vertices

Vertex Processing

final vertex positions
Making Lane-Carpenter match edges
Making Lane-Carpenter match edges

Non-uniform
Making Lane-Carpenter match edges

Non-uniform

??

5
Non-isoparametric splits

DiagSplit: adaptive, crack-free, sub-patch parallel
DiagSplit adapts as well as Lane-Carpenter

7% more vertices

Triangle area relative to target (1/2 pixel triangles)

Too small

Too large

DiagSplit

Lane-Carpenter

[Fisher 09]
DiagSplit: produces better meshes using fewer vertices

Triangle area relative to target (1/2 pixel triangles)

[Direct3D 11 Uniform] [DiagSplit]

40% fewer vertices

[Too small] [Too large]

1/8 1/4 1/2 1 2 4 8x

[Fisher 09]
DiagSplit tessellation pipeline

Base patch data

DiagSplit

Compute Constraints

Surface Eval(u,v)

sub-patches + edge rates

Uniform tessellation (mesh generation)

sub-patch meshes

Vertex Processing

final vertex positions

Fixed-function

Programmable
How to implement parallel split?

- See breath-first approaches that use data-parallel scan
  [Patney 08] [Eisenacher 09] [RenderAnts]

- On a CPU, probably want depth-first
RASTERIZATION:
Parallelizing rasterization across micropolygons (+ analyzing cost of motion and defocus blur)

Fatahalian, Luong, Boulos, Akeley, Mark, Hanrahan (HPG 2009, best paper)
Rasterization

Compute coverage using point-in-triangle tests
Compute “possibly covered” pixels
Data-parallel sample tests

[Seiler 08]
[Greene 96]
[Fuchs 89]
[Pineda 88]
Micropolygons: most point-in-polygon tests fail

61% of candidate samples inside triangle

6% of candidate samples inside triangle

Low sample test efficiency!
# Micropolygon rasterization

<table>
<thead>
<tr>
<th>For each MP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Setup</strong></td>
</tr>
<tr>
<td><strong>Bound</strong></td>
</tr>
<tr>
<td><strong>Test</strong></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
Parallel micropolygon rasterization

Process multiple micropolygons simultaneously

Input micropolygons

Output fragments
## Parallel micropolygon rasterization (MP Rast)

<table>
<thead>
<tr>
<th></th>
<th>PARALLEL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Setup</strong></td>
<td>Cull polygon if back-facing</td>
</tr>
<tr>
<td><strong>Bound</strong></td>
<td>Compute subpixel bbox of MP</td>
</tr>
</tbody>
</table>
| **Test**   | For each sample in bbox  
              Test MP-sample coverage | UTILIZATION? |
How efficient is MP Rast?

- What percentage of tested samples fall within the micropolygon?

- Does parallelization across micropolygons efficiently utilize vector processing?
MP Rast increases sample test efficiency

(2.5 to 6x more efficient than 16 sample block)

- Conventional rasterizer: 16 sample (4x4) block
- MP Rast

<table>
<thead>
<tr>
<th>Sample test efficiency (%)</th>
<th>1 sample / pixel</th>
<th>4 samples / pixel</th>
<th>16 samples / pixel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conventional rasterizer</td>
<td>2%</td>
<td>6%</td>
<td>11%</td>
</tr>
<tr>
<td>MP Rast</td>
<td>12%</td>
<td>20%</td>
<td>28%</td>
</tr>
</tbody>
</table>

(2.5 to 6x more efficient than 16 sample block)
MP Rast sustains high vector utilization

Overall Utilization (%) vs. Vector width (Number of polygons processed in parallel)

- 80% at 8 polygons processed
- 74% at 16 polygons processed
- 67% at 32 polygons processed
- 62% at 64 polygons processed
Micropolygon rasterization is simple, but expensive!

Perform more sample tests
No “all in” cases
Can’t amortize per-triangle work over many tests
Micropolygon rasterization is simple, but expensive!

**Rasterization computation:**
- 1080p resolution, 30 Hz
- 4 samples per pixel
- Simple scene (10 million micropolygons)

**Estimated cost of software GPU implementation:**
- ~1/3 of high-end NVIDIA GPU running CUDA

*Group currently investigating fixed-function micropolygon rasterization*
MOTION BLUR AND CAMERA DEFOCUS

Using stochastic point sampling (not 2D hacks)
Moving micropolygon
X, T plane
Motion blur + defocus: 5D point-in-polygon tests (XY, T, lens UV)
Candidate samples
INTERVAL (4 time intervals)
INTERVAL (4 time intervals)
INTERVAL

small motion = tight bounds
INTERVAL

large motion = loose bounds
## INTERVAL

<table>
<thead>
<tr>
<th>Setup</th>
<th>...</th>
</tr>
</thead>
</table>
| Bound       | For each time interval  
              Compute MP bbox over interval |
| Test        | For each sample in interval and in bbox  
              Position MP at sample T  
              Test MP-sample coverage |
INTERLEAVE: main idea

- Limit the number of unique times (or lens positions) used to sample coverage
Interleaved sampling

[Mitchell 91]
[Keller 01]
INTERLEAVE

For each MP

Setup

...  

Bound

For each unique time T
Position MP at T
Compute MP bbox at T

Test

For each tile in bbox
Test MP-sample coverage
- How much do motion blur and camera defocus cost?

- What is relative performance of INTERVAL, INTERLEAVE under varying amounts of motion or defocus?
Soccer jump

16x multi-sampling

INTERVAL: 16 time intervals
INTERLEAVE: 64 unique times
Enabling motion/defocus blur costs 3 to 7x more

- Point-in-polygon tests are more expensive
- INTERVAL, INTERLEAVE perform more tests than NOBLUR

Sample test efficiency (stationary geometry, perfect focus)

<table>
<thead>
<tr>
<th>Method</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOBLUR</td>
<td>28%</td>
</tr>
<tr>
<td>INTERVAL</td>
<td>11%</td>
</tr>
<tr>
<td>INTERLEAVE</td>
<td>5%</td>
</tr>
</tbody>
</table>
INTERVAL’s performance varies with motion
INTERLEAVE more efficient than INTERVAL at high motion
~30 pixels of motion blur equates performance

Rasterization Cost

Amount of motion blur (pixels)
Extension to motion blur / defocus

- Costs 3 to 7x more in flops than NOBLUR
- INTERVAL more efficient until motion is large
- INTERLEAVE more efficient under high motion, moderate to high defocus
- Both algorithms are inefficient!
  - Only 1 in 20 polygon-sample tests generate hits
  - Can you do better?
SHADING:

Eliminating redundant shading work from the pipeline

Fatahalian, Boulos, Hegarty, Akeley, Mark, Moreton, Hanrahan (SIGGRAPH 2010)
Recall: test coverage multiple times per pixel

multi-sample anti-aliasing [Akeley 93]
Texture data is pre-filtered to avoid aliasing
(one shade per pixel is sufficient)
Texture data is pre-filtered to avoid aliasing

(one shade per pixel is sufficient)
Quad-fragments (2x2 pixel blocks)

Subtract neighboring texture coordinates for derivatives

\((s_{00}, t_{00})\)  \(\rightarrow\)  \((s_{10}, t_{10})\)  \(\rightarrow\)  \((s_{11}, t_{11})\)
Shaded quad-fragments
Pixels along edges shaded multiple times

Shading computations per pixel
Surfaces with micropolygons shaded $> 8 \times$ per pixel

Current GPU: shading computations per pixel

- 25 pixel triangles
- 15 pixel triangles
- 5 pixel triangles
- 1 pixel triangles

Color scale:
8 - 7 - 6 - 5 - 4 - 3 - 2 - 1
REYES: shade once per vertex

Color per micropolygon vertex

Final pixels

[Cook 87]
Fragment shading is still compelling

- **Smooth evolution from GPU status quo**
  - Fragment shading supported by GPUs today
  - High-quality shading of larger polygons

- **Naturally avoids shading what is not seen**
  - Off-screen or occluded surfaces

*Also see [Ragan-Kelley et al.]*
GPU quad-fragment shading pipeline

Triangle mesh
(from tessellation)

Rasterization

Quad-fragments

Shading
Redundant shading!

quad-fragment from tri 1

quad-fragment from tri 2

final pixels
Merge quad-fragments from different triangles

Rasterization

Merge-buffer (quad-fragments)

Merge

Merged quad-fragments for shading

Shading
Challenge: avoiding merges that introduce image artifacts
Edges must be anti-aliased
Current GPU
Naive merging

Key idea: only merge quad-fragments from adjacent triangles
Current GPU shading (no merging)

1/2 pixel triangles
Merging: 8.5x less shading

1/2 pixel triangles
Merging has benefit for “small” triangles

Reduces shading by $2\times$ for 10 pixel area “small” triangles
Near identical visual quality

Quad-fragment merging

Current GPU (no merging)
Near identical visual quality

Quad-fragment merging

Current GPU (no merging)
Merging reduces shading work

Average reduction: 8.2x

Shaded quad-fragments by current GPU (relative to quad-fragment merging)
Frog

Rough surface has many edges: fewer opportunities to merge
Recap

- Quad-fragment merging reduces micropolygon over-shade
  - shade surfaces (not triangles) at a density of once per pixel

- Not a radical change to rasterization or shading
  - uses quad-fragments for derivatives
  - compatible with edge anti-aliasing
  - supports shading large triangles

- Shaded point counts are comparable to vertex shading
A micropolygon rendering pipeline

Increased efficiency by:

- adaptive tessellation,
- re-optimizing rasterization,
- and eliminating redundant shading