A Real-time Micropolygon Rendering Pipeline

Kayvon Fatahalian
Stanford University
Detailed surfaces

Rendering goals

Highly detailed surfaces: micropolygons
(But interoperate with large triangle rendering in same frame)

Accurate camera defocus and motion blur

[Near future] real-time system
It is inefficient to render micropolygons using the current graphics pipeline.
How does the real-time graphics pipeline evolve to enable efficient micropolygon rendering?

How should surfaces be tessellated into micropolygons?
How can micropolygons be rasterized efficiently?
How expensive is motion blur and defocus?
Should the pipeline shade like GPUs or like REYES?

(changes to algorithms, abstractions, and HW are fair game)
TESSELLATION:
How do we add adaptive tessellation to the pipeline?
Generate micropolygons on demand

Base primitives

Smooth tessellation

Final displaced surface
D3D tessellates parametric surfaces

Coarse Vertex Processing

Primitive Processing

Tessellate

Fine Vertex Processing

Rasterize+Cull

Fragment Processing

Frame buffer ops

Compute tessellation factor

Generate new triangles

Compute position of new triangle verts

New in D3D11

Fixed function =
Need adaptive tessellation

- Uniform tessellation per patch over/under tessellates
Performing good adaptive tessellation is very important.

Complexity pays for itself.

- Less surface evaluation
- Less shading
- Less (and more efficient) rasterization
Split-dice adaptive tessellation

Split: recursively divide primitive (adaptive)
Dice: simple uniform tessellation (fast)
D3D11 tessellation

- Don’t split, just dice
  - Tessellation factors for primitive edges
  - “Fancy” dicer stitches edges to uniform interior

REYES dicing

D3D11 dicing
Parallel, adaptive tessellation must avoid cracks
Offline techniques suboptimal for real-time

- Retain the entire micropolygon mesh and “stitch” adjacent subpatches
  - Very robust
  - Hard to parallelize (can’t stream)

- Use only power-of-two dicing factors
  - Easier to parallelizable
  - T-junctions
  - Over tessellates
Split-dice for a real-time pipeline

- **DiagSplit**: simple solution
  - Adaptive tessellation
  - Independent subpatch processing
  - Crack-free
  - No power-of-two constraints

Fisher et al. [Siggraph Asia 2009]
DiagSplit: idea 1

- Leverage D3D11 tessellator as advanced dicer
  - Edge-based tessellation factor function: \( h(\text{edge}) \)
  - Let \( h \) evaluate displacement

\[
egin{array}{ccc}
  h=5 & h=4 & h=6 \\
  h=SPLIT & h=4 & h=SPLIT \\
  h=3 & h=3 & \\
  \text{Dicable: send to Dicer} & \text{Evaluate } h \text{ on subpatches}
\end{array}
\]
DiagSplit: idea 2

- Scale area of internal polygons to hit target area precisely

Can scale tessellation of interior region without creating cracks
**DiagSplit: idea 3**

- Permit “diagonal” (non-isoparametric) splits
What is the cost of splitting?

- Most time is spent evaluating final surface positions
- Then you’ve got to rasterize and shade
- Avoiding over tessellation pays for itself
- 10 million MP/sec on one core of Intel I7
Parallel tessellation is an active area

- **Expressibility of parametric surfaces is increasing**
  - Approximate Catmull-Clark surfaces [Loop (TOG 2008)]
  - Extension to creases and corners [Kovacs (I3D 2009)]

- **Variety of implementations**
  - Depth-first (cache efficient, tiny footprint)
  - Breath-first subdivision (fits current GPU compute model well)
    - Patney (SIGGRAPH Asia 2008), Patney (HPG 2009)
    - Eisenacher (I3D 2009)
    - RenderAnts
  - CUDATess [Schwarz (Eurographics 2009)]
D3D11 Pipeline

- Coarse Vertex Processing
- Primitive Processing
- Tessellate (Dice)
- Fine Vertex Processing

Micropolygon Pipeline

- Coarse Vertex Processing
- Primitive Processing
- Split
- Tessellate (Dice)
- Fine Vertex Processing
- Tessellation Rate Func
- Bounding Box Func

Programmable

Fixed function
RASTERIZATION:
How efficiently can micropolygons be rasterized? (and what’s the cost?)
Step 1: per-polygon preprocessing (setup)

- Clip, back face cull, compute edge equations
- Make point-in-polygon tests cheap
Step 2: compute candidate sample set

- Coarse reject/accept of samples

Coarse uniform grid
Step 2: compute candidate sample set

- Coarse reject/accept of samples

Coarse uniform grid

Hierarchical descent
Step 2: compute candidate sample set

- Coarse reject/accept of samples

Coarse uniform grid

Hierarchical descent
Step 2: compute candidate sample set

- Coarse reject/accept of samples

Coarse uniform grid

Hierarchical descent
Step 3: point-in-polygon tests

- Test “stamp” of samples against polygon simultaneously (data-parallel)

[Pineda 88]
[Fuchs 89]
[Greene 96]
[Seiler 08]
Micropolygons: more polygons = more setup
Micropolygons: coarse reject not useful
Micropolygons: large stamps yield low efficiency

47% of tested samples inside triangle

6% of tested samples inside triangle
Re-implement rasterizer

- Rasterize many micropolygons simultaneously

(Fatahalian et al. HPG 2009)
Reimplement rast to increase efficiency
(2.5 to 6x more efficient)

Conventional rasterizer:
- 16 sample (4x4) stamp
- MICROPOLYGON-PARALLEL

Sample test efficiency (%)

- No multi-sampling: 2%
- 4x multi-sampling: 6%
- 16x multi-sampling: 28%

(2.5 to 6x more efficient)
Micropolygon rasterization is expensive

Primary visibility computation:

- 1080p resolution, 30 Hz
- 4x multi-sampling
- Simple scene (10 M micropolygons)

Estimated cost of GPU SW implementation:

- Approximately 1/3 of high-end GPU

Fixed-function micropolygon rasterization is appealing
MOTION BLUR AND DEFOCUS:
How expensive are motion blur/defocus?
Motion blur and defocus

- Many 2D-techniques for approximating blur
  [Sung 02]
  [Demers 04]

- Stochastic point sampling
  [Cook 84, Cook 86]
  [Akenine-Moller 07]
Moving micropolygon
Soccer jump

16x multi-sampling

64 unique time samples
Enabling motion/defocus blur costs 3 to 7x more

- Point-in-polygon tests are more expensive
- Algorithms that support blur perform more point-in-polygon tests

Stationary geometry, perfect focus:
Tests 2.5x more samples
Cost 3x more
Performance varies with motion
Most efficient algorithm depends on scene conditions
[Pixar method] costs increase sharply with defocus
**D3D11 Pipeline**

- Coarse Vertex Processing
- Primitive Processing
- Tessellate (Dice)
- Fine Vertex Processing
- Rasterize+Cull

**Micropolygon Pipeline**

- Coarse Vertex Processing
- Primitive Processing
- Split
- Tessellate (Dice)
- Tesselatte (Dice)
- Fine Vertex Processing
- Rasterize (5D)

**Functions**

- Tessellation Rate Func
- Bounding Box Func

**Shutter Duration, Lens Parameters**

- Open and close shutter
- Vertex positions

**Options**

- Unchanged =
- Programmable =
- Fixed function =
SHADING:
Should a micropolygon pipeline shade like today’s GPUs or like REYES?
GPUs shade micropolygons inefficiently

At least 4x overshade!
(must share shading results across micropolygons)
MSAA implementations break down with blur

Motion-blurred polygon input

Covered samples are spread widely across screen

Rasterize

(inefficient to represent as hit mask)
Future fragment shading?

Decoupled sampling: Ragan-Kelley et al.

Rasterize

Visibility sample hits

Shading Result Cache

Shading requests

Shaded samples

Fragment processing

Fragments

Frame Buffer Ops
REYES-style shading

- Perform shading computations at vertices
- Requires connectivity for derivatives
  - Not a problem: system has the diced mesh
- Decoupled shading for motion blur and defocus “just works”
Overshade in a “REYES” system

- Overtessellation
- Conservative, coarse occlusion culling
Shading system design space

- **Current GPU fragment shading**
  - Shade only what’s visible
    - Fine early Z
    - Deferred shading

- Sample in screen space

- Sample in object space

- Shade before exact visibility known

- REYES: shade at diced grid vertices
SUMMARY
Good adaptive tessellation is very important

DiagSplit: Build split on top of D3D11 Tess

Abstract SPLIT as non-programmable stage

Micropolygon Pipeline

- Coarse Vertex Processing
- Primitive Processing
- Split
- Tessellate (Dice)
- Fine Vertex Processing
- Fine Vertex Shading
- Rasterize (5D)
- Future Fragment Proc.
- Frame buffer ops

Tessellation Rate Func
Bounding Box Func
Shading on surface: Need derivatives & good tessellation

We've got the diced grid for connectivity

**Micropolygon Pipeline**

- Coarse Vertex Processing
- Primitive Processing
- Split
- Tessellate (Dice)
- Fine Vertex Evaluation
- **Fine Vertex Shading**
- Rasterize (5D)
- Future Fragment Proc.
- Frame buffer ops

Tessellation Rate Func
Bounding Box Func

(requires mesh connectivity)
**Micropolygon Pipeline**

- **Rasterization:** re-implement for micropolygons
- **Fixed-function HW likely**
- **Point sampling for motion blur and defocus are 3-7x more costly**

**Steps:**

1. **Coarse Vertex Processing**
2. **Primitive Processing**
3. **Split**
4. **Tessellate (Dice)**
5. **Tessellation Rate Func**
6. **Bounding Box Func**
7. **Fine Vertex Processing**
8. **Fine Vertex Shading**
9. **Rasterize (5D)**
10. **Future Fragment Proc.**
11. **Frame buffer ops**

- **shutter duration, lens parameters**
Tessellate (Dice)
Rasterize (5D)

Micropolygon Pipeline

Must share shading across micropolygons to avoid overcompute
(quad per MP is inefficient)

Motion blur/defocus change fragment shading (even without micropolygons)

Coarse Vertex Processing
Primitive Processing
Split
Tessellate (Dice)
Fine Vertex Processing
Fine Vertex Shading
Rasterize (5D)
Future Fragment Proc.
Frame buffer ops

Tessellation Rate Func
Bounding Box Func
It’s coming!

- We’ll have the flops from future HW
- Lot’s of active research on algorithms
- Real-time micropolygon rendering will be possible in the future
Thank you