Parallel View-Dependent Tessellation of Catmull-Clark Subdivision Surfaces

Anjul Patney¹  Mohamed S. Ebeida²  John D. Owens¹

University of California, Davis ¹  Carnegie Mellon University ²
Smooth Surfaces in Interactive Graphics

Motivation
Resolution independent higher-order surfaces in real-time graphics

Challenges
• Fast GPU-friendly tessellation
• Fine-grained view-adaptivity
• Cracks and Pinholes
Representing Smooth Surfaces

**Polygonal Meshes**
- Lack View-Adaptivity
- Inefficient storage/transfer

**Parametric Surfaces**
- Collections of smooth patches
- Hard to animate

**Subdivision Surfaces**
- Widely popular, ease of modeling
- Flexible Animation

[Boubekeur and Schlick 2007]
Existing Work

Subdivision Surfaces in real-time graphics

• CPU-based tessellation [Bolz and Schröder 2002]
  • Frequent CPU-GPU data transfer
  • Fixed maximum levels of subdivision

• Subdivision using graphics shaders [Shiu et al. 2005], [Bunnell 2005]
  • Requires patching the input mesh
  • Limited view-adaptivity

• Ray tracing of subdivision surfaces [Benthin et al. 2007]
  • Requires patching the input mesh
  • Coarse view-adaptivity
Existing Work

Programmable GPU Tessellation

• Bézier Patches
  [Patney and Owens 2008], [Eisenacher et al. 2009]
  • Parametric surfaces
  • Cracks and T-junctions

• CUDA Tessellation Framework
  [Schwarz and Stamminger 2009]
  • Parametric surfaces only
Existing Work

Hardware Tessellation

• DirectX 11 Dedicated Tessellation Units
  • Primarily parametric surfaces

• Approximate Catmull-Clark Surfaces [Loop and Schaefer 2008]
  • Lost C¹ continuity, require separate normals
  • Inexact surfaces
This Paper: Contributions

A parallel approach to Catmull-Clark subdivision

- Highly parallel, GPU-friendly
- Simple, robust data management
- Dynamic view-dependence
- No cracks or T-junctions
Review: Catmull-Clark Subdivision

- Given: An arbitrary base mesh $M_0$
- Recursively refine to produce $M_1$, $M_2$, $M_3$ ...
  - Fast convergence to a smooth surface
Refinement Procedure

Face Points
Refinement Procedure

Edge Points
Refinement Procedure

Vertex Points
Refinement Procedure

Subdivided Mesh
Basic Approach

- Basic Approach
- Subdivision Test
- Adaptive Mesh Refinement
- OpenGL
- GPU
- Intermediate mesh
- Input mesh (coarse)
Basic Approach

GPU

Intermediate mesh

Input mesh (coarse)

GenVP
Parallel

GenEP
Parallel

GenFP
Parallel

Subdivision Test

OpenGL
Challenges

- Data structure choice
  - Maintaining mesh structure
  - Efficiency on graphics hardware

- View-Dependence
  - Spatial (along the surface)
  - Temporal (dynamic viewpoint)

- Cracks and T-junctions
  - Avoid visual artifacts
  - Should not degrade performance
Data Structure

• Mesh Representation Goals
  • Easily exposed parallelism
  • Minimal communication
  • Low storage overhead
  • Straightforward rendering

• Problems with common Mesh Representations
  • Serialized pointer-accesses
  • Variable-size elements
  • Often cannot be rendered directly
Our Choice

• Maintain three arrays
  • **VertexBuffer**, with vertex = \{x, y, z, valence\}
  • **FaceBuffer**, with face = \{v_0, v_1, v_2, v_3\}
  • **EdgeBuffer**, with edge = \{v_0, v_1, f_0, f_1\}

• Justification
  • Parallelism expressed at all three levels
  • Fixed-size elements
  • Straightforward rendering
  • Low storage overhead
  • Complexity of updates?
• Scattered atomic add (face-point)
• Scattered atomic add (edge midpoint)
View Dependence
Subdivision Test

Faces

Edges

Vertices

Parallel Prefix-Sum
[Sengupta et al. 2007]
Varying Subdivision Criteria

Screen-space extent
Varying Subdivision Criteria

Surface Orientation
Varying Subdivision Criteria

Curvature
Varying Subdivision Criteria

Silhouette-enhancement
Cracks

Uniform Refinement  Adaptive Refinement
T-Junctions

Multisampling disabled
Crack-fix as a post-process
Incrementally Resolving Cracks

Quad-only Refinement
Incrementally Resolving Cracks

2-Refinement Templates
[Schneiders 96]

![Diagram showing 2-refinement templates and their application on a grid]
Incrementally Resolving Cracks

“Active” Vertices

• Placed at alternating transition vertices

• Help choosing a subdivision template
Incrementally Resolving Cracks

Tagging Active Vertices

- Initialized potential tags with the base mesh
- Convert to active at transition
- Apply templates
- Update potential tags
Incrementally Resolving Cracks

Tagging Active Vertices

• Initialized potential tags with the base mesh
• Convert to active at transition
• Apply templates
• Update potential tags
Example
Implementation

Hardware Platform
• NVIDIA GeForce GTX 280

Computing Architecture
• NVIDIA CUDA 2.2

Subdivision criterion used for results
• Screen-space extent
Results
Results
Results – Performance Behavior

- Linear with output complexity
- Performance approaches 2.5–3M faces/sec
Summary

- Parallel GPU tessellation of Catmull-Clark surfaces
- Robust data management for subdivision
- Dynamically view-dependent
- Fixing cracks in parallel

Support for mesh boundaries and textures
Limitations

• Quad-only meshes
  • Bloats off-line storage and transfer

• Crack-fix affects the 1-ring neighborhood
  • Aggressive subdivision can help
  • Rarely a problem in practice

• Memory access patterns
  • Several non-coalesced dependent lookups
  • Implementation limited by memory bandwidth
Future Work

• Extension to alternate refinement schemes
  • Loop
  • Doo-Sabin

• Efficient memory management
  • Programmable geometry caching
Acknowledgments

Feedback and Suggestions
• Eric Lengyel
• Anonymous paper reviewers

Models
• Bay Raitt (Valve Software)
• Headus Inc.

Funding and Equipment
• NSF Award 0541448
• SciDAC Institute for Ultrascale Visualization
• NVIDIA
Thanks!
EXTRA SLIDES
Breadth-first Vs Depth-first Subdivision

- One thread for one face
  - Limited parallelism

Initial number of faces = 6

Initial number of faces = 1450
Calculating vertex normals

• Two methods

1. Averaged normals of adjacent faces
   a. Regular mean
   b. Weighted mean

2. Subdivision shading
Calculating vertex normals
Frame time division

![Graph showing the fraction of frame time spent in subdivision vs mesh size (faces)]
Results – Performance Scaling

![Graph showing performance scaling with mesh size](image)

- **bigguy**
- **killeroo**
- **monsterfrog2**

Frame Time (ms) vs Mesh Size (faces)
Results – Performance Scaling

![Graph showing performance scaling across different screen sizes and models.]

- **bigguy**
- **complex**
- **killeroo**
- **monsterfrog2**
- **threeblock**
Results – Performance Scaling

The graph above illustrates the performance scaling of different models as a function of the subdivision criterion (pixels). The x-axis represents the subdivision criterion, while the y-axis shows the subdivision performance in Mfaces/s. Each line color and marker correspond to different models:

- bigguy (blue circles)
- complex (green squares)
- killeroo (red triangles)
- monsterfrog2 (purple crosses)
- threeblock (orange X)

As the subdivision criterion increases, the performance (Mfaces/s) generally increases for all models, indicating improved efficiency with higher refinement levels.
Varying subdivision depth

![Graph showing subdivision performance vs depth](image)

- **Subdivision Performance (Mfaces/s)** on the y-axis.
- **Subdivision Depth (levels)** on the x-axis.
- Different symbols and colors represent different models:
  - bigguy
  - complex
  - killeroo
  - monsterfrog2
  - threeblock
Need for silhouette enhancement