Impostors for Interactive Parallel Computer Graphics

Orion Sky Lawlor

olawlor@acm.org

2004/11/29

http://charm.cs.uiuc.edu/users/olawlor/academic/thesis/

Overview

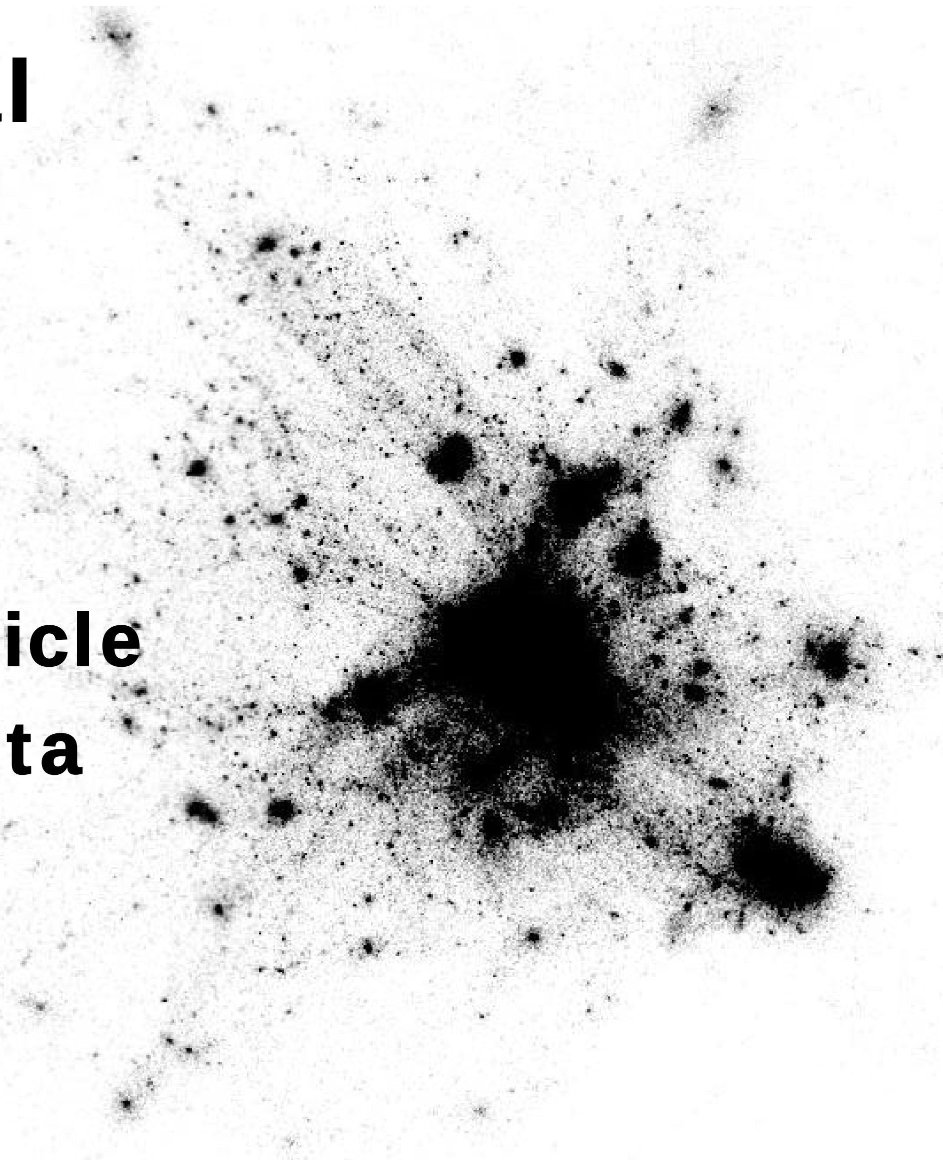
- Case Studies
- Prior Work
  - Serial Rendering and Problems
  - Parallel Rendering and Problems
  - Impostors
- New Work
  - Parallel Impostors Technique
  - Better Rendering Enabled by Parallel Impostors
- Conclusions

Selection of Case Studies

- Current state of the art hardware and techniques can handle simple small smooth surfaces well
  - Small in both meters and bytes
  - Smooth; low in geometric complexity
    - But possibly high in (theoretical) polygon count
  - Simple lighting
  - Simple aliased point-sampled geometry
- Large, complex geometry not handled well
  - Large in bytes and meters
  - Geometric complexity
  - Rendering fidelity
  - Rendering complexity

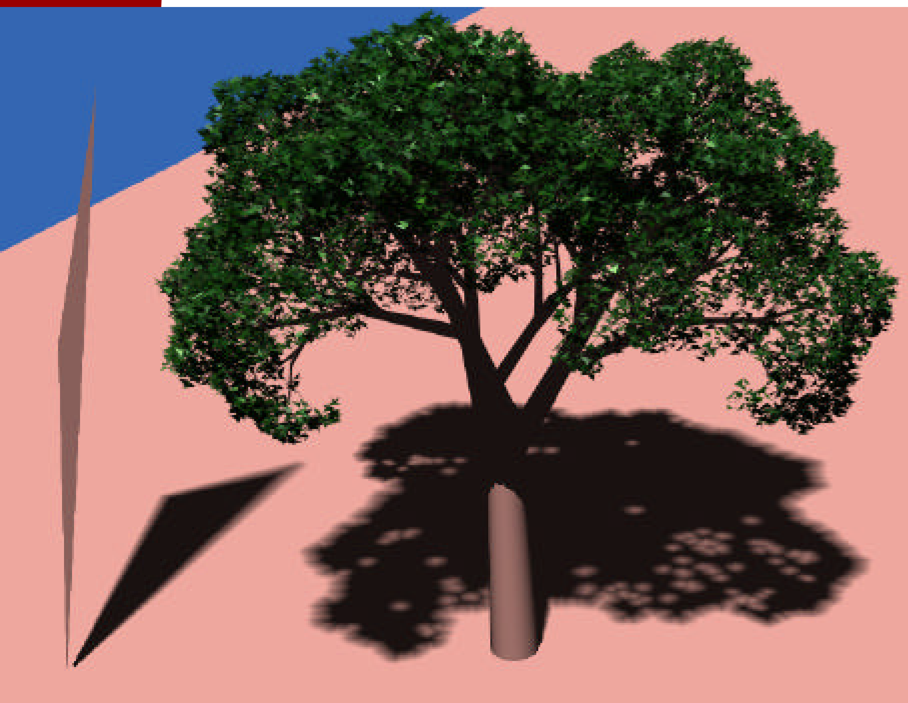
Large Particle Dataset

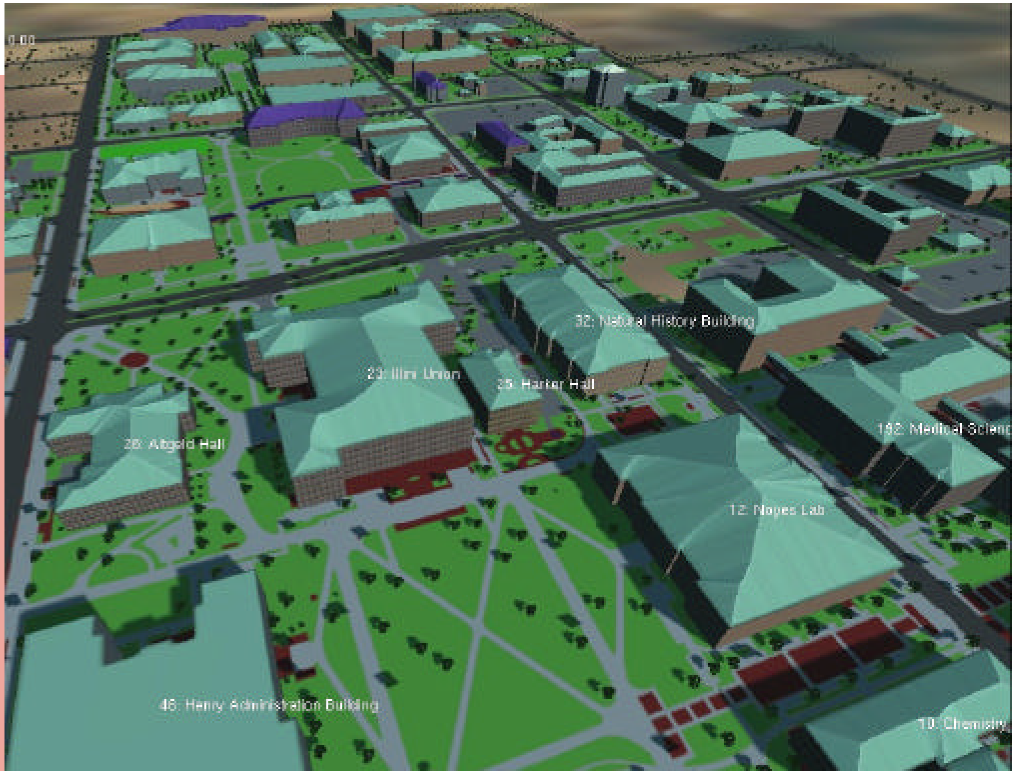
- **Computational Cosmology Dataset**
- **Large size**
  - **50M particles**
  - **20 bytes/particle**
  - **=> 1 GB of data**



Campus Dataset

- **Large virtual world**
- **Built on a terrain model**
- **Complex rendering**
  - **Light, shadow, geometric detail**





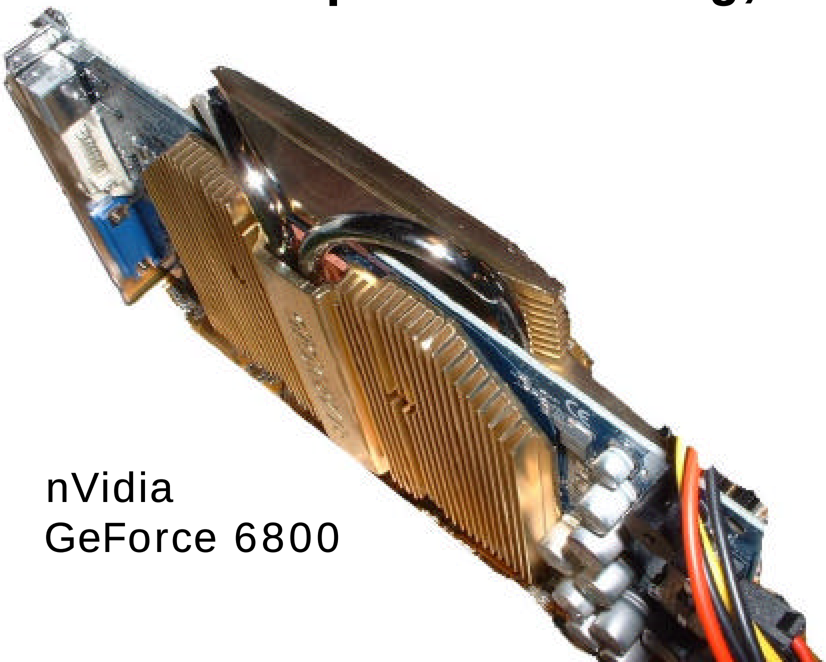
Prior Approaches and Unsolved Problems

Approach #1:

Just use a good graphics card!

Approach #1: Serial Rendering

- **Graphics cards are fast, right?**
  - **So just render everything on the graphics card**
- **Exponentially Increasing Performance**
  - **Consumer hardware vertex processing (1999)**
  - **Programmable hardware pixel shaders (2001)**
  - **Hardware floating-point pixel processing (2003)**
  - **Per-pixel branching, looping, reads/writes (2005)**



nVidia
GeForce 6800

- **Draws *only* polygons, lines, and points**
- **Supports image <u>texture mapping</u>, transparent blending, primitive lighting**

Graphics Card Performance

Triangle Setup
Projection, lighting, clipping, ...

→

Pixel Rendering
Texturing, blending

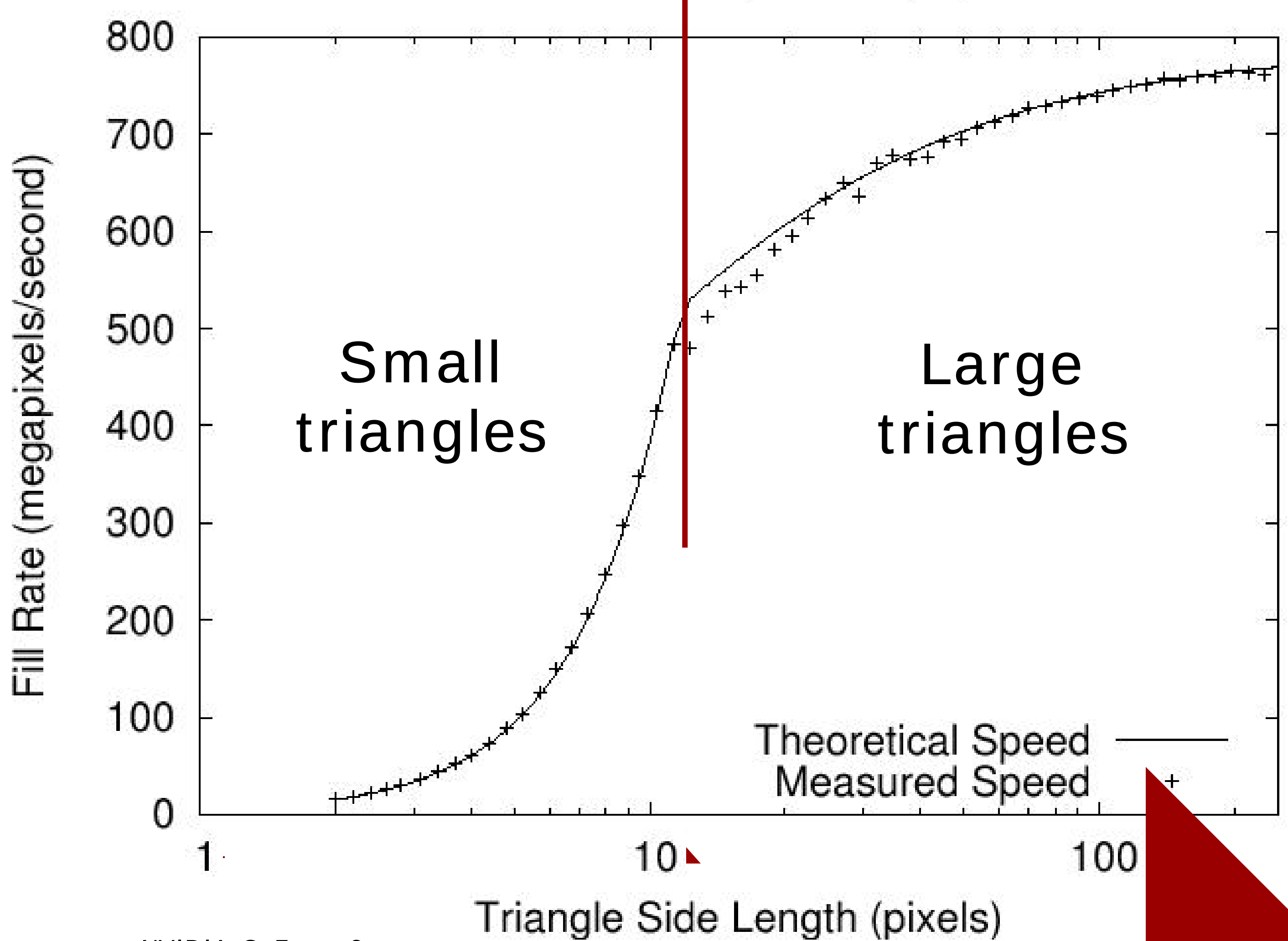
$$t = \max(\alpha, \beta(s + \gamma r))$$

t	total time to draw triangle (seconds)
a	triangle setup time (about 50ns/triangle)
b	pixel rendering time (about 1ns/pixel)
s	area of triangle (pixels)
r	rows in triangle
g	pixel cost per row (about 3 pixels/row)

!

Graphics Card: Usable Fill Rate

$$B_C = \frac{s}{t} = \min(\frac{s}{\alpha}, \frac{1}{\beta(1+\gamma r/s)})$$



NVIDIA GeForce 3

Smooth vs Complex Surfaces

- **Smooth Surfaces**
  - **Polygons/patches**
  - **Continuous, well-defined surface**
  - **Lots of occlusion**
  - **Mesh simplification [Garland 97]**
  - **Can sometimes be made fillrate limited**

- **Complex Surfaces**
  - **Particles/splats**
  - **All discontinuity; <u>no</u> well-defined surface**
  - **Not much occlusion**
  - **Lazy surface expansion [Hart 93]**
  - **<u>Never</u> fillrate limited**

Serial Rendering Drawbacks

- **Graphics cards are fast**
  - **But not at rendering lots of tiny geometry:**
    - **50K polygons/frame OK**
    - **50M pixels/frame OK**
    - **50M polygons/frame not OK**
- **Problems with complex geometry do not utilize current graphics hardware well**
- **The techniques we will describe can improve performance for geometry-limited problems**

Approach #2:
Just use a parallel machine!

Approach #2: Parallel Rendering

- **Parallel Machines are fast, right?**
  - **Scale up to handle huge datasets**
  - **Render lots of geometry simultaneously**
  - **Send resulting images to client machine**
- **Tons of raytracers [John Stone’s Tachyon], radiosity solvers [Stuttard 95], volume visualization [Lacroute 96], etc**
- **“Write an MPI raytracer” is a homework assignment**
- **Movie visual effects studios use frame-parallel offline rendering (“render farm”)**
- **CSAR Rocketeer Apollo / Houston: frame parallel**
- **Offline rendering basically a <u>solved</u> problem**

Parallel Rendering Advantages

- **Multiple processors can render geometry simultaneously**

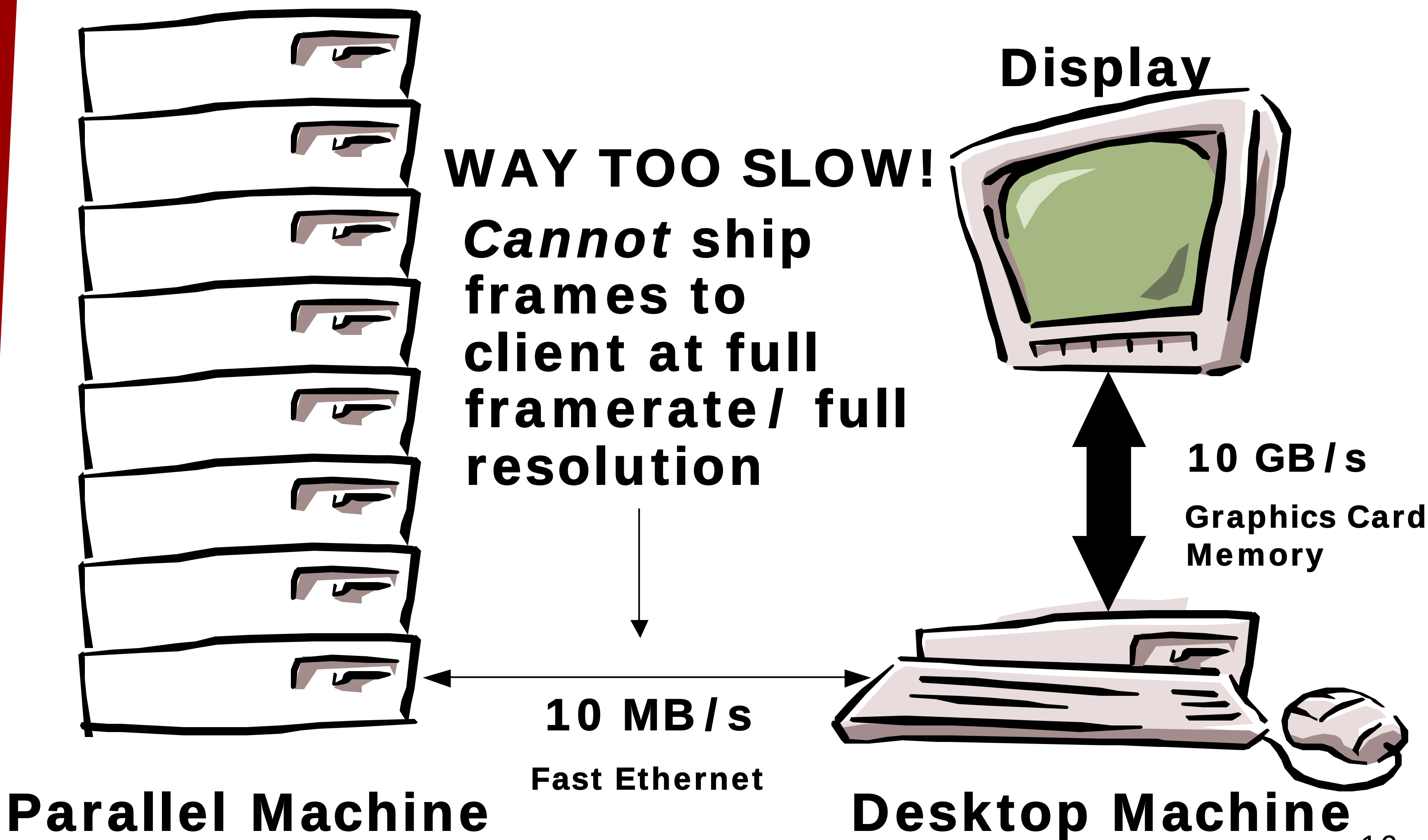
Processors	4	8	16	24	32	48
MParticles/second	7.14	15.71	32.71	49.18	65.49	81.68

48 nodes of Hal cluster: 2-way 550MHz Pentium III nodes connected with fast ethernet

- **Achieved rendering speedup for large particle dataset**
- **Can store huge datasets in memory**
- **Ignores cost of shipping images to client**

Parallel Rendering Disadvantage

- **Link to client is too slow!**

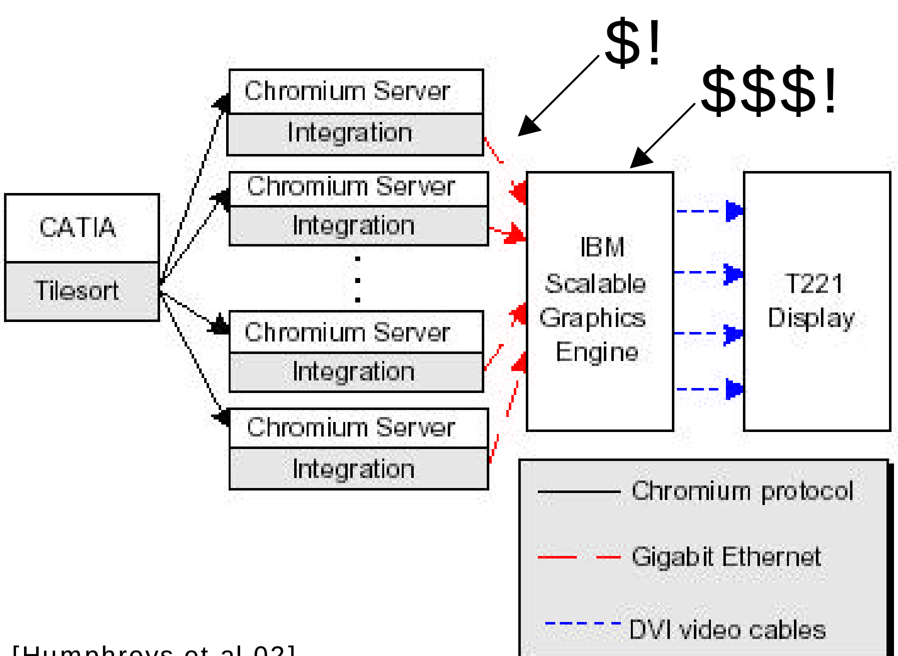


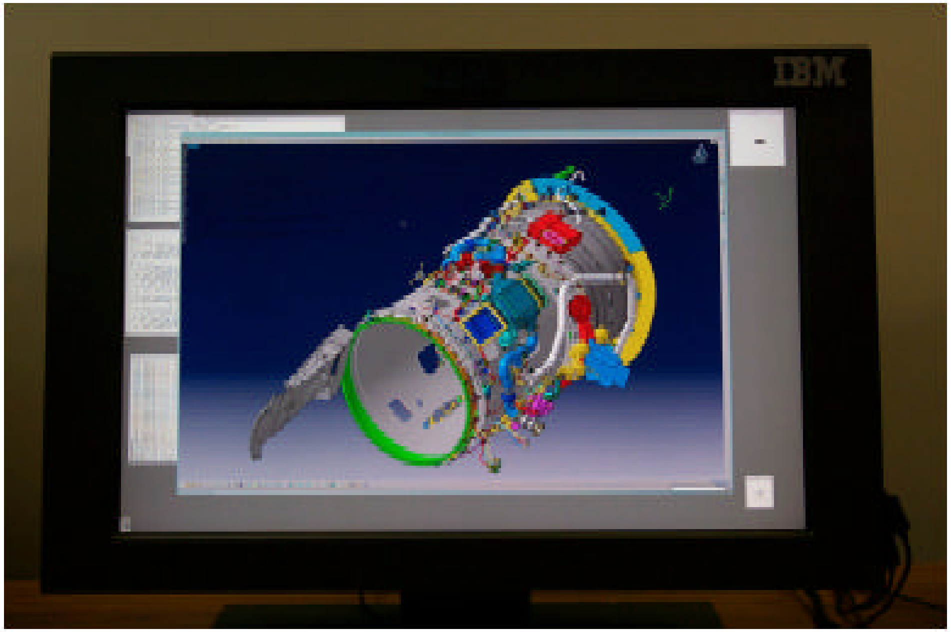
Parallel Rendering Bottom Line

- Conventional parallel rendering works great offline
- But not for interactive rendering
  - Link to client has inadequate bandwidth
    - Can’t send whole screen every frame
  - System has zero latency tolerance
    - Client has nothing to do but wait for next frame
    - If parallel machine hiccups, client drops frames
- The techniques we will describe can improve parallel rendering bandwidth usage and provide latency tolerance

Parallel Rendering in Practice

- **Humphreys et al's Chromium (aka Stanford's WireGL)**
  - **Binary-compatible OpenGL shared library**
  - **Routes OpenGL commands across processors efficiently**
  - **Flexible routing--arbitrary processing possible**
  - **Typical usage: parallel geometry generation, screen-space divided parallel rendering**
- **Big limitation: screen image reassembly bandwidth**
  - **Need multi-pipe custom image assembly hardware on front end**





[Humphreys et al 02]

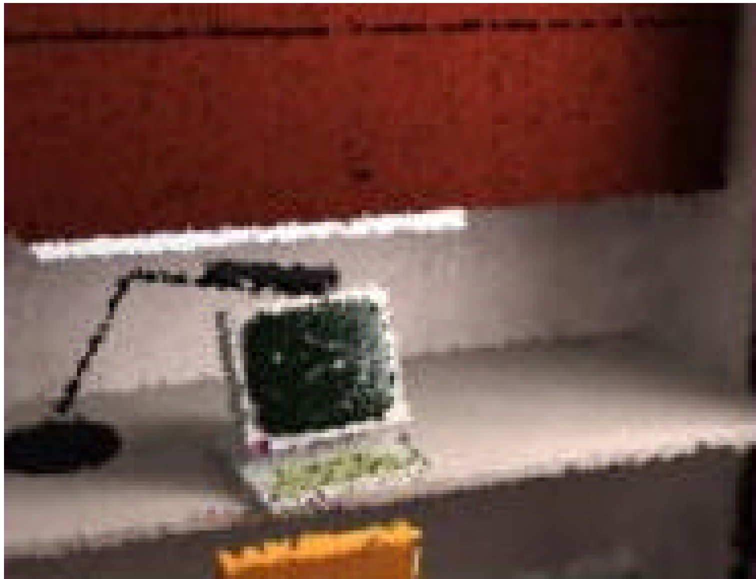
Unconventional Parallel Rendering

- **Bill Mark's post-render warping**
  - **Parallel server sends every N'th frame to client**
  - **Client interpolates remaining frames by warping server frames according to depth**

[Mark 99]

[Ward 99]

- **Greg Ward's "ray cache"**
  - **Parallel Radiance server renders and sends bundles of rays to client**
  - **Client interpolates available nearby rays to form image**



Impostors

Fundamentals

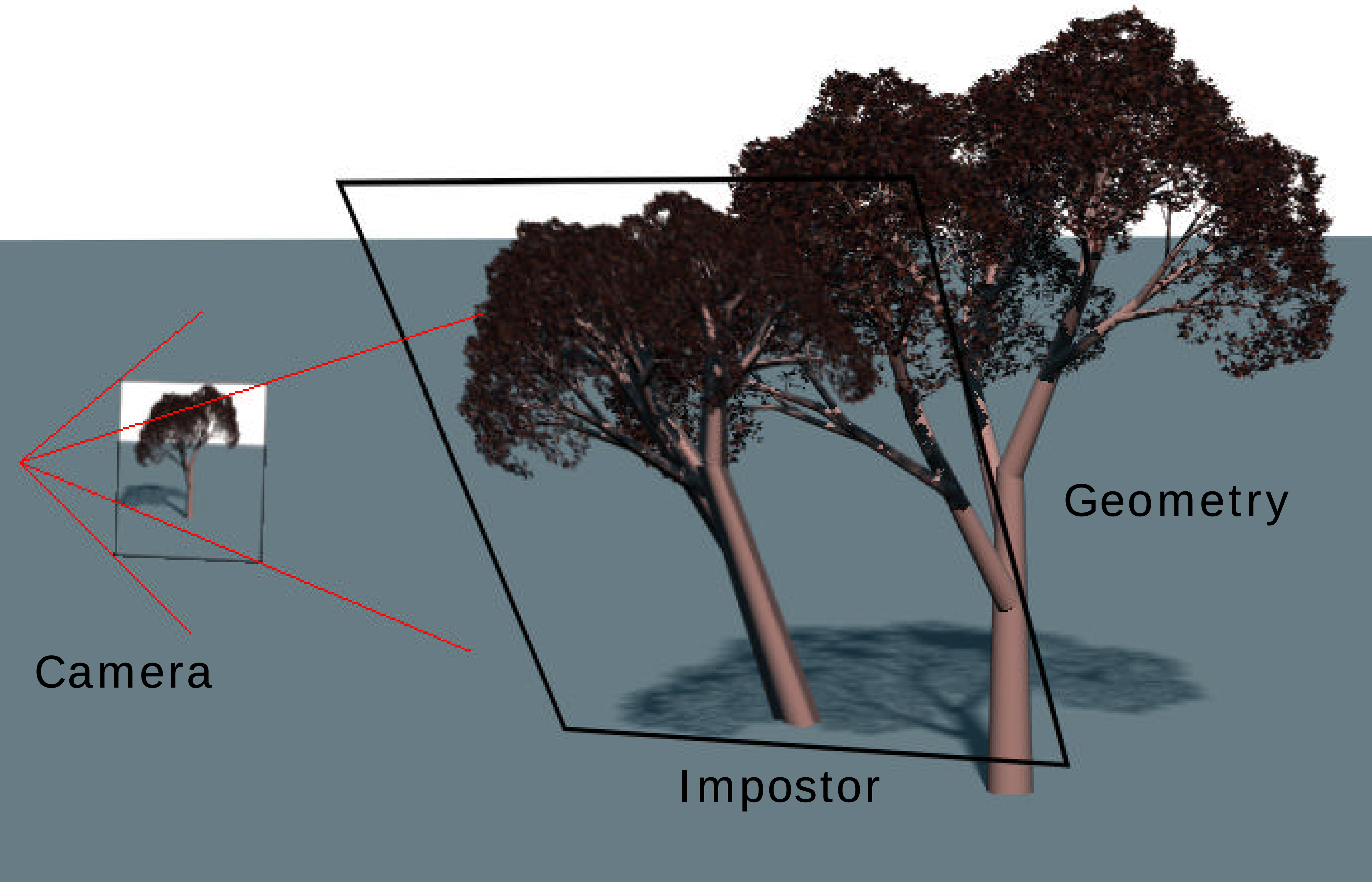
Prior Work

Impostors

- Replace 3D geometry with a 2D image
  - Image an "impostor"
- 2D image fools viewer into thinking 3D geometry is still there
- Prior work
  - Pompeii murals
  - *Trompe l'oeil* ("trick of the eye") painting style
  - Theater / movie backdrops
- Main Limitation
  - No parallax-- must update impostor as view changes

[Harnett 1886]

Impostors : Idea



Impostor Reuse

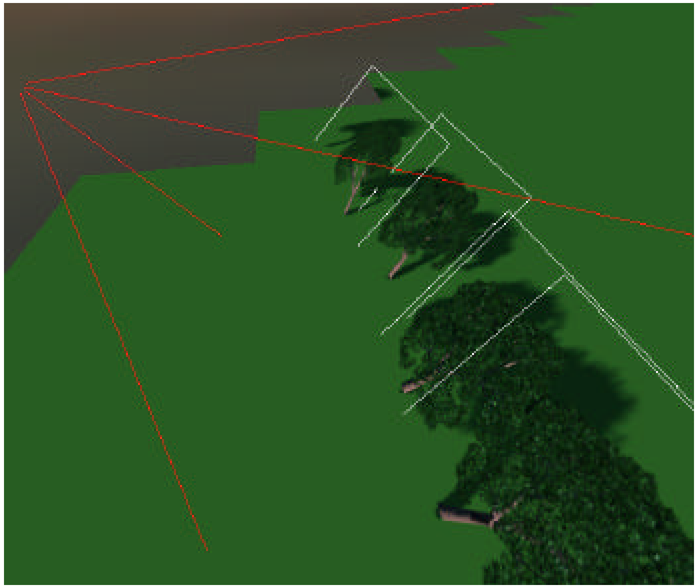
- We don't need to redraw the impostors every frame
  - If we did, impostors wouldn't help!
- Can <u>reuse</u> impostors from frame to frame
  - Can reuse forever under camera rotation
- Far away or flat impostors can be reused many times
  - Assuming reasonable camera motion rate

	$d = 0.05$	$d = 0.25$	$d = 1$	$d = 5$
$z = 1$	1	1	1	1
$z = 5$	10	2	1	1
$z = 25$	263	52	12	2
$z = 100$	4216	841	208	40

Number of frames impostor can be reused, for various depth ranges (columns) and distances (rows)

Impostors for Complex Scenes

- Use different impostors for different objects in scene
  - Get some parallax even without updating
- Number of impostors can depend on viewpoint



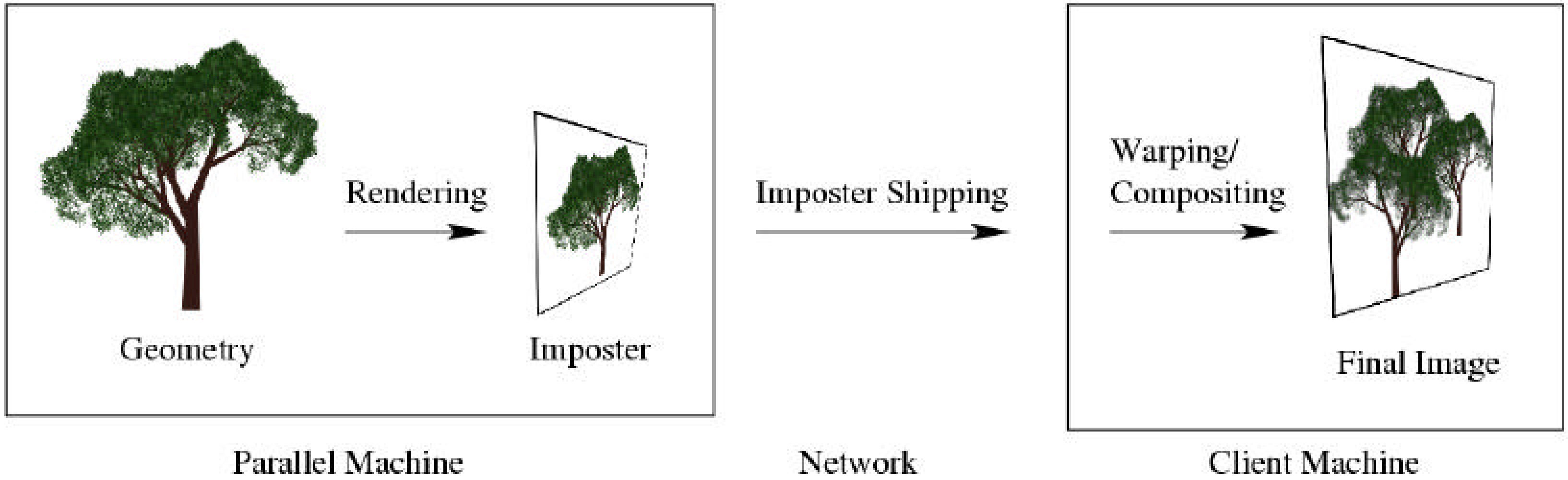
Parallel Impostors

Our Proposed Solution

Parallel Impostors Technique

- **Key observation: impostor images don't depend on one another**
- **So render impostors in parallel!**
  - **Uses the speed and memory of the parallel machine**
    - **Fine grained-- lots of potential parallelism**
  - **Geometry is partitioned by impostors**
    - **No "shared model" assumption**
- **Reassemble world on serial client**
  - **Uses rendering bandwidth of client graphics card**
  - **Impostor reuse cuts required network bandwidth to client**
    - **Only update images when necessary**
  - **Impostors provide latency tolerance**

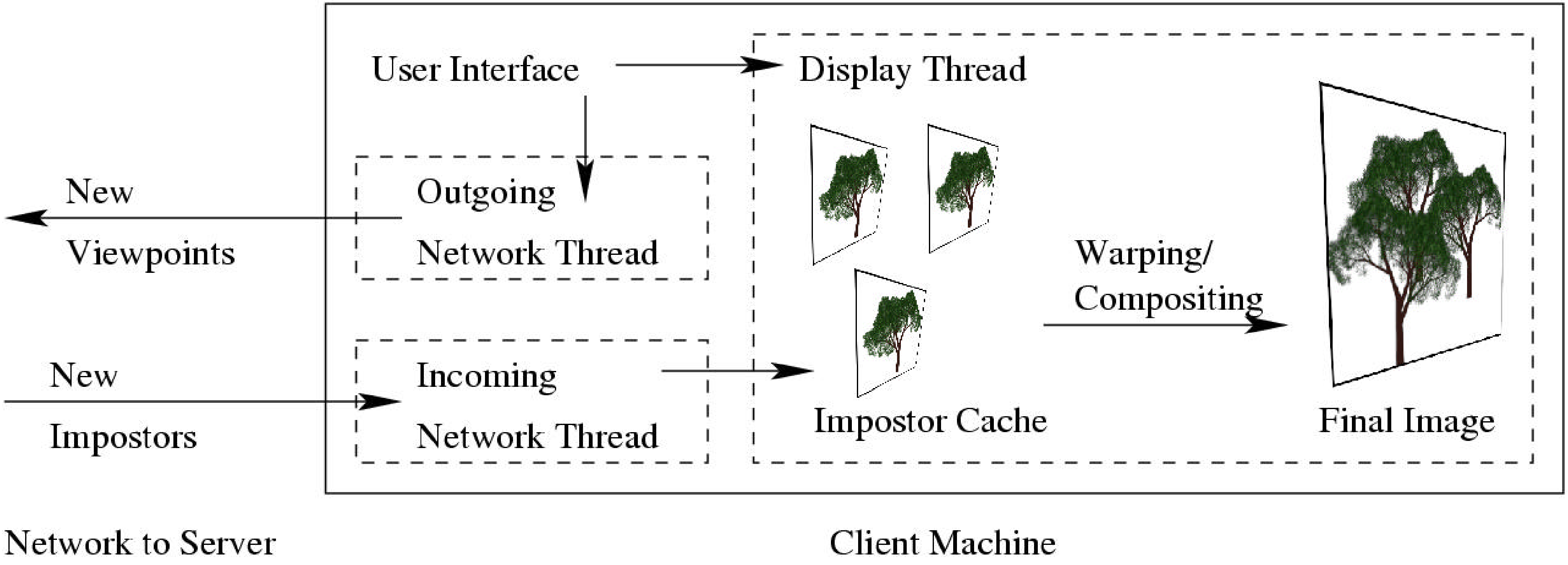
Client/Server Architecture



- **Parallel machine can be anywhere on network**
  - **Keeps the problem geometry**
  - **Renders and ships new impostors as needed**
- **Impostors shipped using TCP/IP sockets**
  - **CCS & PUP protocol [Jyothi and Lawlor 04]**
  - **Works over NAT/firewalled networks**
- **Client sits on user's desk**
  - **Sends server new viewpoints**
  - **Receives and displays new impostors**

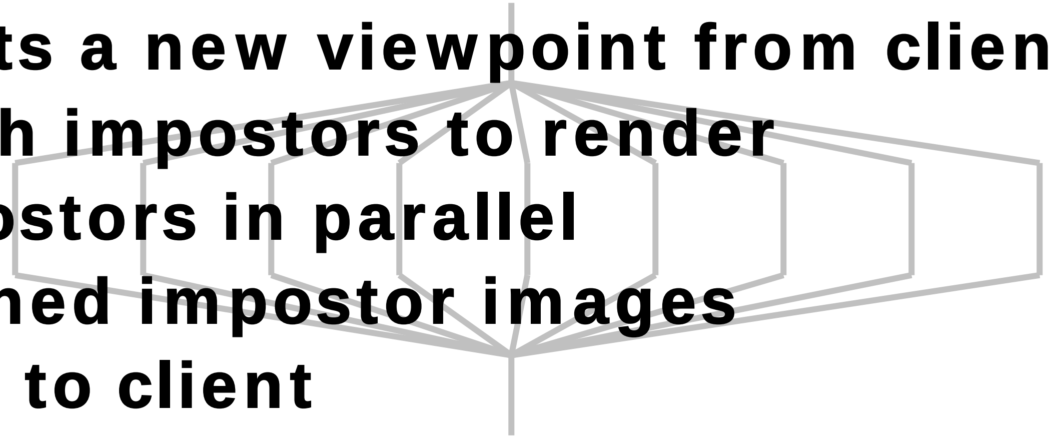
Client Architecture

- **Latency tolerance: client *never* waits for server**
  - **Displays existing impostors at fixed framerate**
    - **Even if they're out of date**
  - **Prefers spatial error (due to out of date impostor) to temporal error (due to dropped frames)**
- **Implementation uses OpenGL for display**
  - **Two separate kernel threads for network handling**



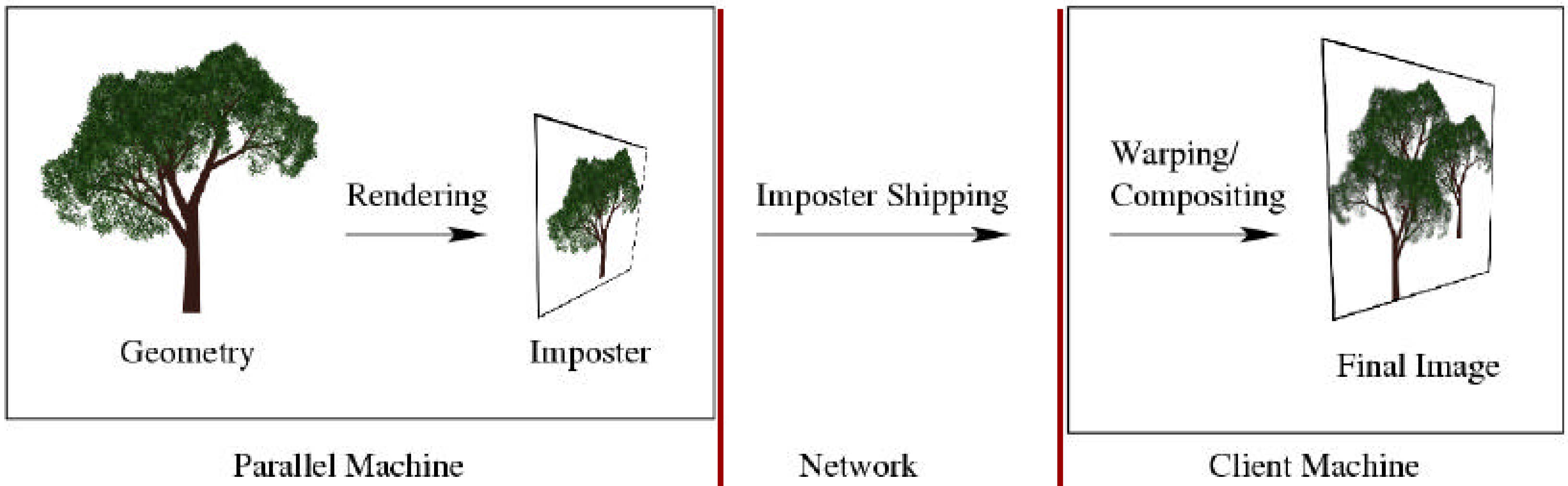
Server Architecture

- Server accepts a new viewpoint from client
- Decides which impostors to render
- Renders impostors in parallel
- Collects finished impostor images
- Ships images to client



- Implementation uses Charm++ parallel runtime
  - Different phases all run *at once*
    - Overlaps everything, to avoid synchronization
    - Trivial in Charm; virtually impossible in MPI
  - Geometry represented by efficient migrateable objects called <u>array elements</u> [Lawlor and Kale 02]
  - Geometry rendered in priority order
  - Create/destroy array elements as impostor geometry is split/merged

Architecture Analysis



$$B = \min(B_R P R,\ B_N C_N R,\ B_C)$$

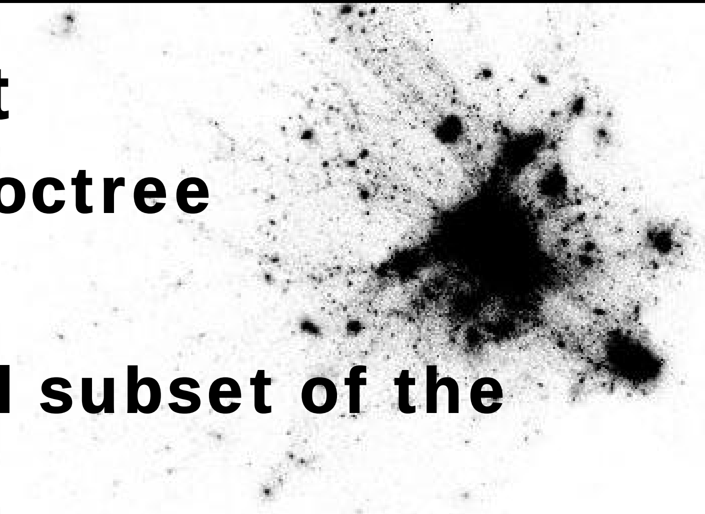
Benefit from Parallelism

Benefit from Impostors

B	Delivered bandwidth (e.g., 300Mpixels/s)
B_R	Rendering bandwidth per processor (e.g., 1Mpixels/s/cpu)
P	Parallel speedup (e.g., 30 effective cpus)
R	Number of frames impostors are reused (e.g., 10 reuses)
B_N	Network bandwidth (e.g., 60 Mbytes/s)
C_N	Network compression rate (e.g., 0.5 pixels/byte)
B_C	Client rendering bandwidth (e.g., 300Mpixels/s)

Parallel Impostors Examples

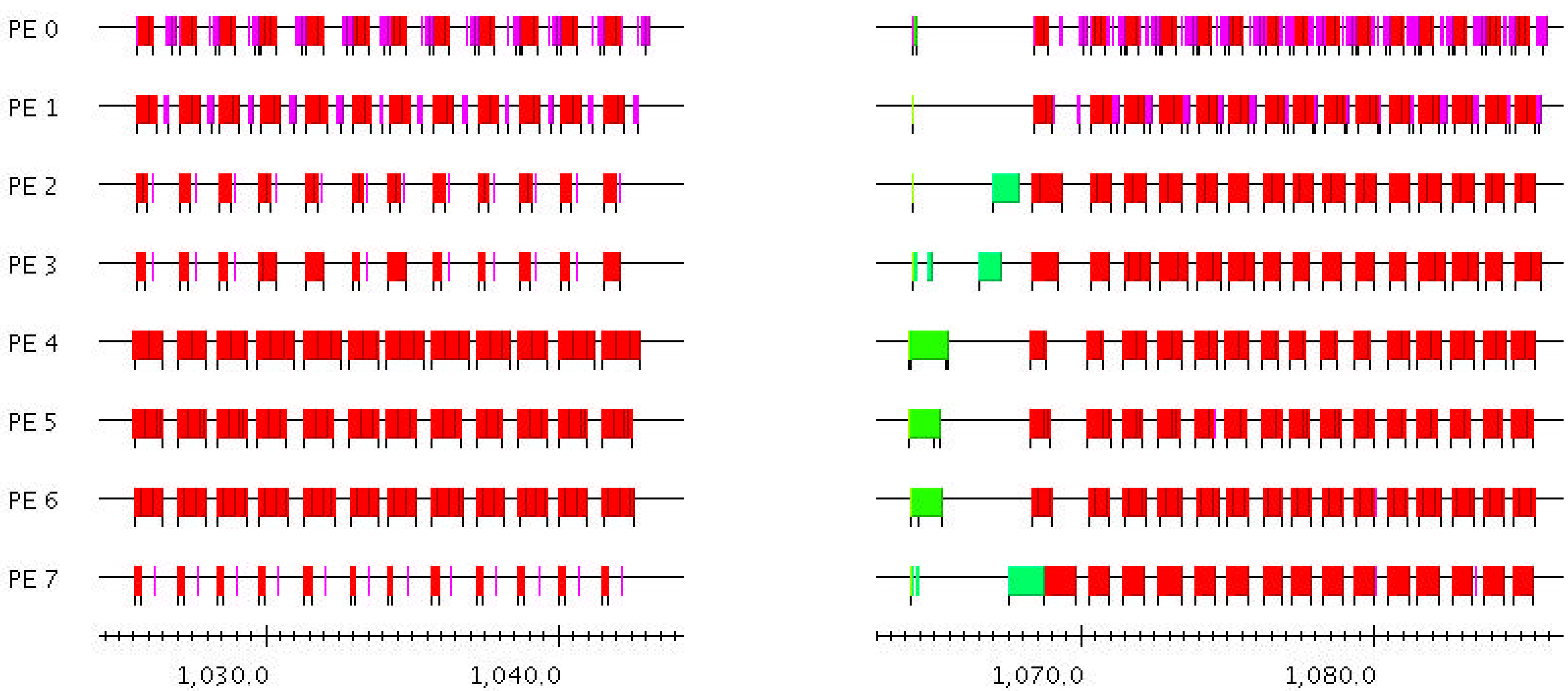
Parallel Particle Example



- Large particle dataset
  - Decomposed using an octree
- Each octree leaf is:
  - Responsible for a small subset of the particles
  - Represented on server by one parallel array element
  - Rendered into an impostor by its array element
    - When the old impostor cannot be reused
  - Drawn on client as a separate impostor
  - Able to migrate between processors for load balance

Parallel Particle Load Balancing

- **Array elements can migrate between processors [Lawlor 03] for load balance**
- **Integrated with Charm++ automated load measurement and balancing system**



Before Balancing

Balancing   After Balancing

Parallel Impostors Performance

- **Parallel Impostors has high framerate and low L^2 error**

Processors	4	8	16	24	32	48
Framerate	61.864 fps	68.742 fps	65.628 fps	62.731 fps	63.993 fps	64.828 fps
Error	0.182655	0.139420	0.127121	0.127309	0.125537	0.135257

48 nodes of Hal cluster: 2-way 550MHz Pentium III nodes connected with fast ethernet

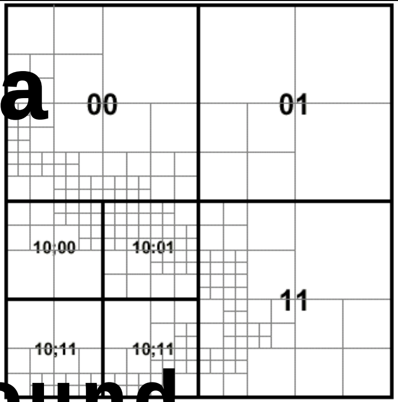
- **Conventional screen shipping has low framerate and high L^2 error**

Processors	4	8	16	24	32	48
Framerate	0.170 fps	0.285 fps	0.458 fps	0.543 fps	0.589 fps	0.681 fps
Error	0.568708	0.482714	0.420433	0.400707	0.388897	0.371073

Parallel Campus Example: Server

- Large terrain model decorated with geometry
- For example, each tree is
  - Represented by one array element
  - Rendered by that array element
    - Only when onscreen and
    - Only when old impostor cannot be reused (based on quality criteria)
  - Able to migrate between processors for load balance

Parallel Campus Example: Server



- Terrain ground texture is a dynamic quadtree
- Each quadtree leaf
  - Represents one patch of ground
  - Stores outlines of sidewalk, roads, grass, brick, etc. on ground
  - Is represented by one array element
    - Using array element bitvector indexing
  - Renders an impostor ground texture for client as needed
  - Divides into children if higher resolution is needed
    - Creating new array elements

Parallel Campus Example: Client

- **Client traverses terrain model decorated with impostors**
  - **Draws terrain and impostors in back-to-front order**
  - **Does not expand offscreen parts of model (checks bounds at each step)**
- **Client can always draw some approximation of scene**
  - **Latency (and latency variation) hiding**

New Features Enabled by Parallel Impostors

Parallel Impostors Enables...

- **Only reason to do any of this is to make <u>new</u> things possible**
- **Showed how very large scenes can now be rendered**
  - **1 GB particle dataset**
- **Can now also do better rendering**
  - **Fully antialiased geometry**
  - **More accurate lighting**
  - **Bigger more realistic databases**

Antialiasing Impostors

Antialiasing Textures

Antialiasing Geometry

Antialiasing Summary

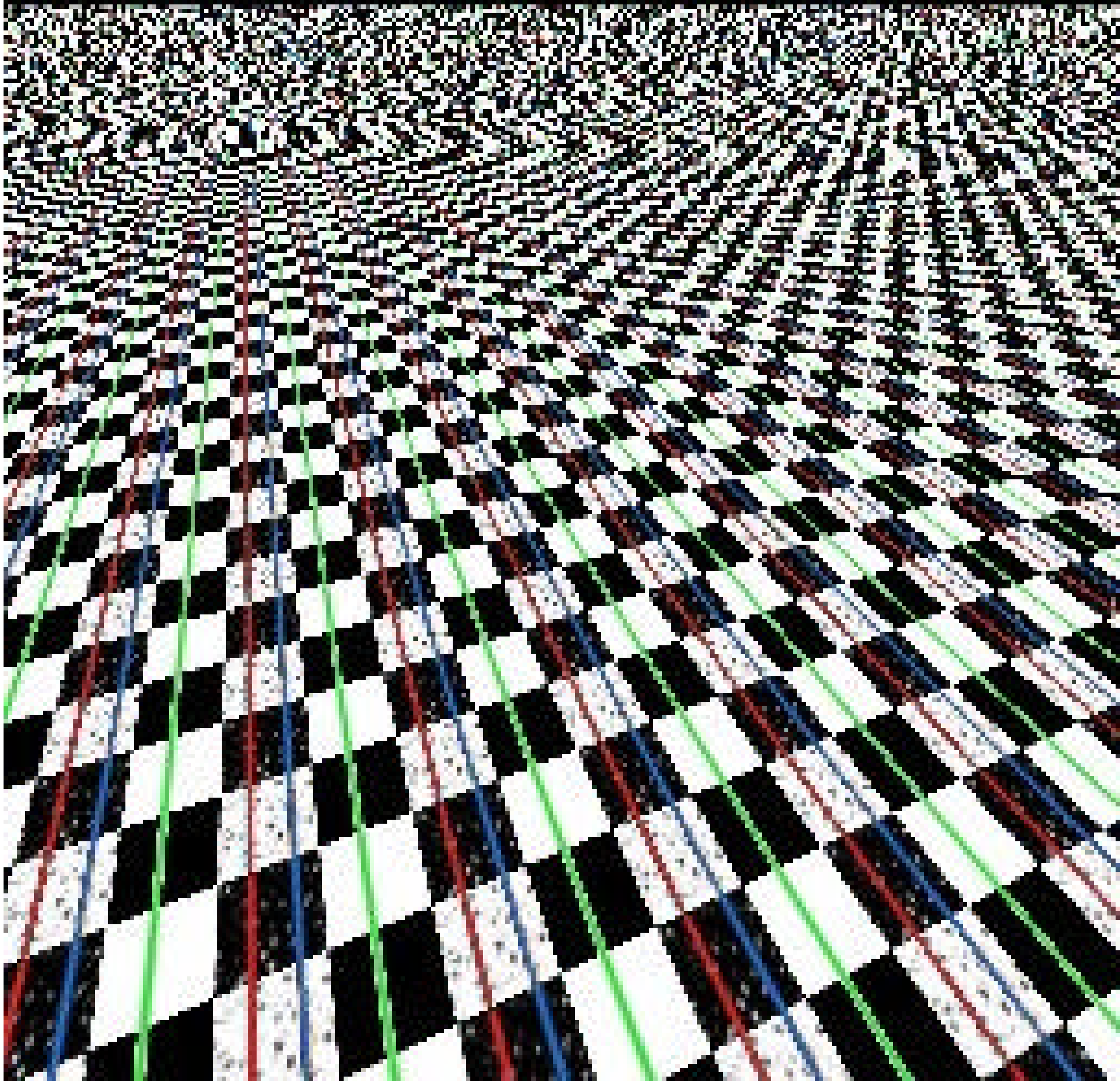
- Textures are easy to antialias
  - Hardware can do it easily
- Geometry is harder to antialias
  - Hardware can't do it easily today
- Impostors turn geometry into texture, but still must antialias geometry
  - Can use any existing antialiasing method

Aliasing: The Problem

Point sampling leads to "aliasing"

Tiny sub-pixel features show up (alias) as noise or large features

The texture on this infinite plane is sampled using the nearest pixel

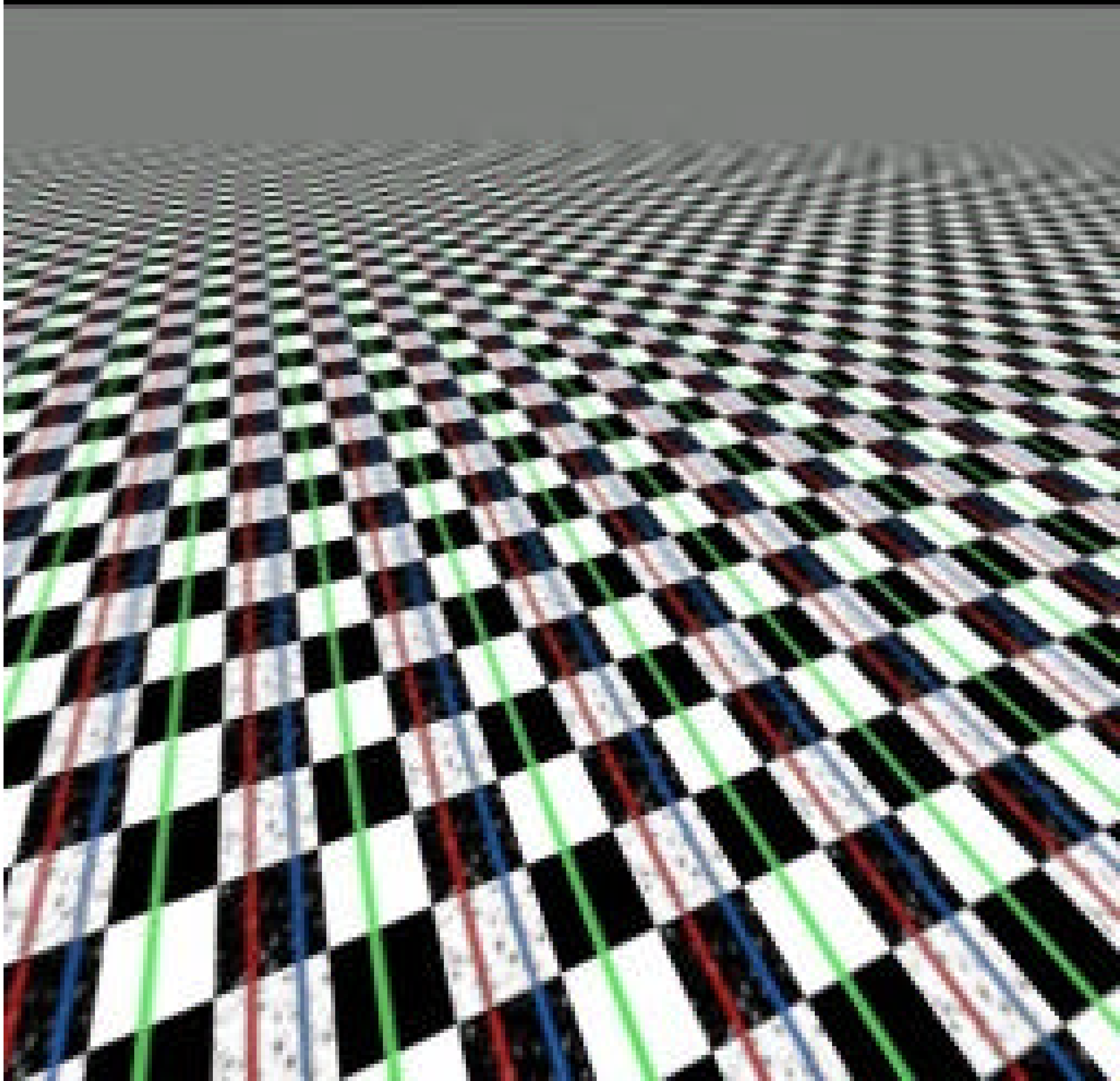


Texture Antialiasing via Mipmaps

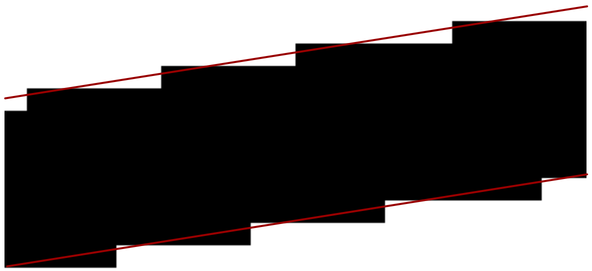
Mipmapping [Williams 83] keeps a pyramid of coarser images, and selects a coarse enough image to eliminate aliases

This coarsening works, but causes excess blurring on tilted surfaces

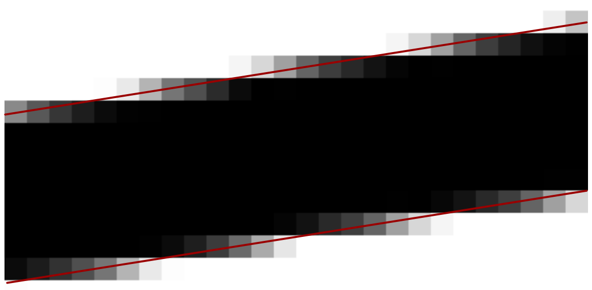
Mipmapping is implemented on all modern graphics hardware



Geometry Antialiasing



Aliased point samples



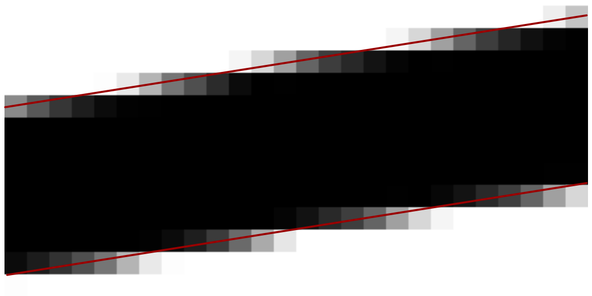
Antialiased filtering

- **Like texture pixels, objects can cover only part of a pixel**
  - **E.g., for tiny objects**
  - **Or along object boundaries**
- **Prior Work:**
  - **Ignore partial coverage and point sample (standard!)**
  - **Oversample and average**
    - **Graphics hardware: FSAA**
    - **Not theoretically correct; close**
  - **Random point samples**
    - **[Cook, Porter, Carpenter 84]**
    - **Needs a *lot* of samples:**

$$s' = \frac{s}{\sqrt{n}}$$

  - **Use analytic technique**
    - **Trapezoids**
    - **Circles [Amanatides 84]**
    - **Polynomial splines [McCool 95]**
    - **Procedures [Carr & Hart 99]**

Geometry Antialiasing via Texture



Antialiased Impostor

- **Texture map filtering is mature**
  - **Very fast on graphics hardware**
  - **Bilinear interpolation for nearby textures**
  - **Mipmaps for distant textures**
  - **Anisotropic filtering becoming available**
  - **Works well with alpha channel transparency**

    [Haeberli & Segal 93]

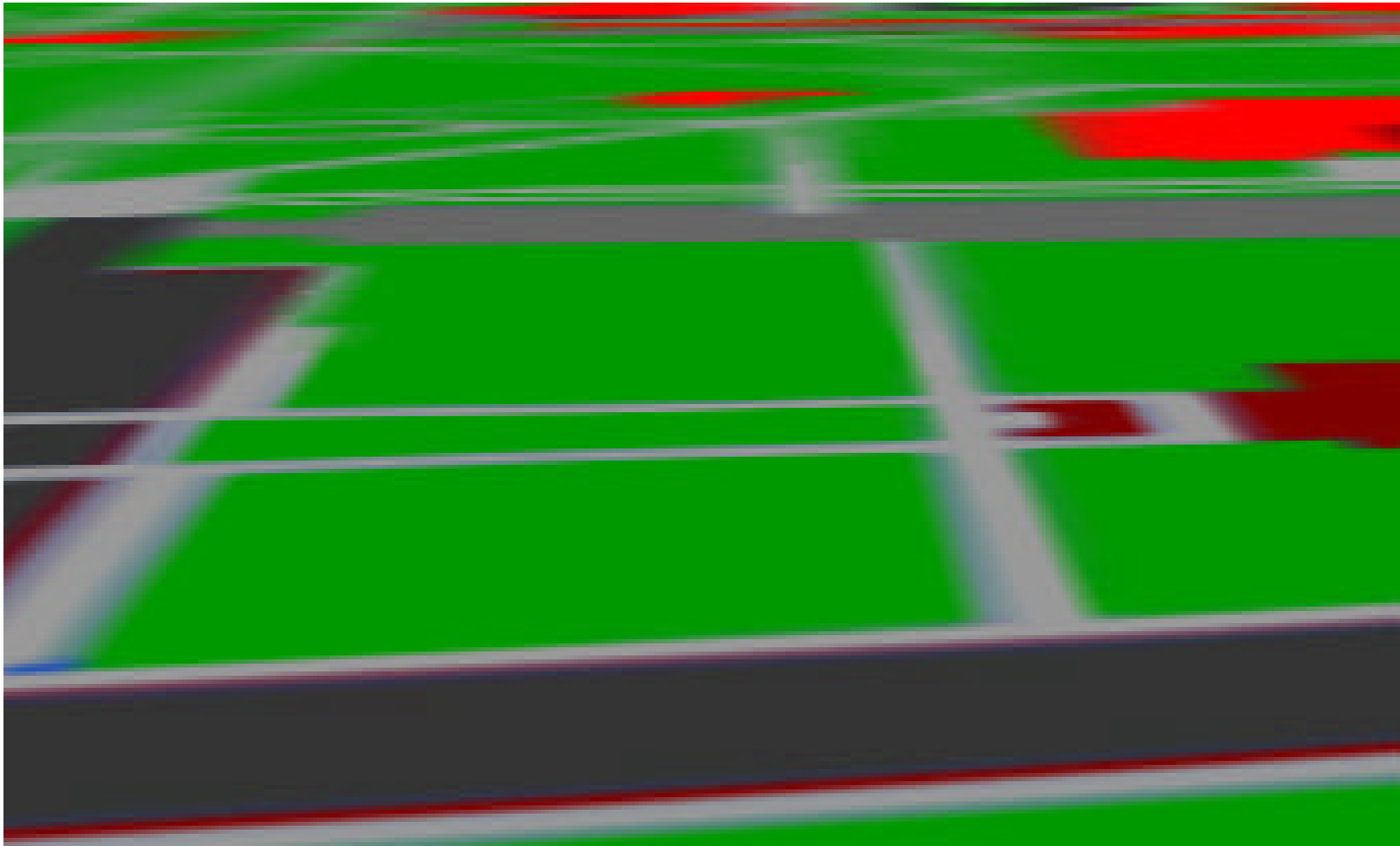
- **Impostors let us use texture map filtering on *geometry***
  - **Antialiased edges**
  - **Mipmapped distant geometry**
  - **Substantial improvement over ordinary polygon rendering**

Antialiased Impostor Challenges

- **Must generate antialiased impostors to start with**
  - **Just pushes antialiasing up one level**
  - **Can use any antialiasing technique. We use:**
    - **Trapezoid-based integration**
    - **Blended splats**
- **Must render with transparency**
  - **Not compatible with Z-buffer**
  - **Painter's algorithm:**
    - **Draw from back-to-front**
    - **A radix sort works well**
    - **For terrain, can avoid sort by traversing terrain properly**

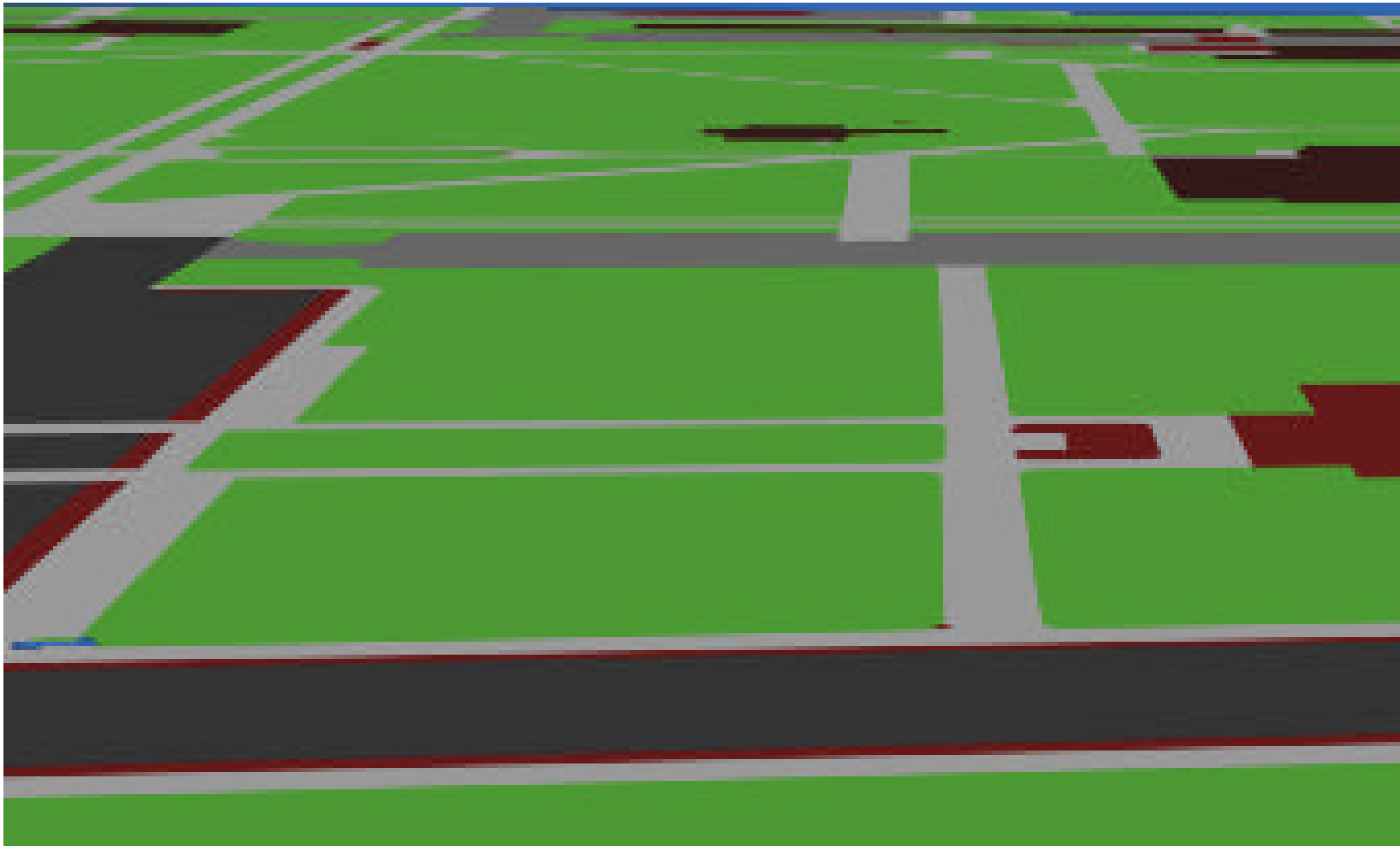
Ground Texture Antialiasing

- Campus example, ground as simple texture
- Mipmaps are fast, but cause excessive blurring



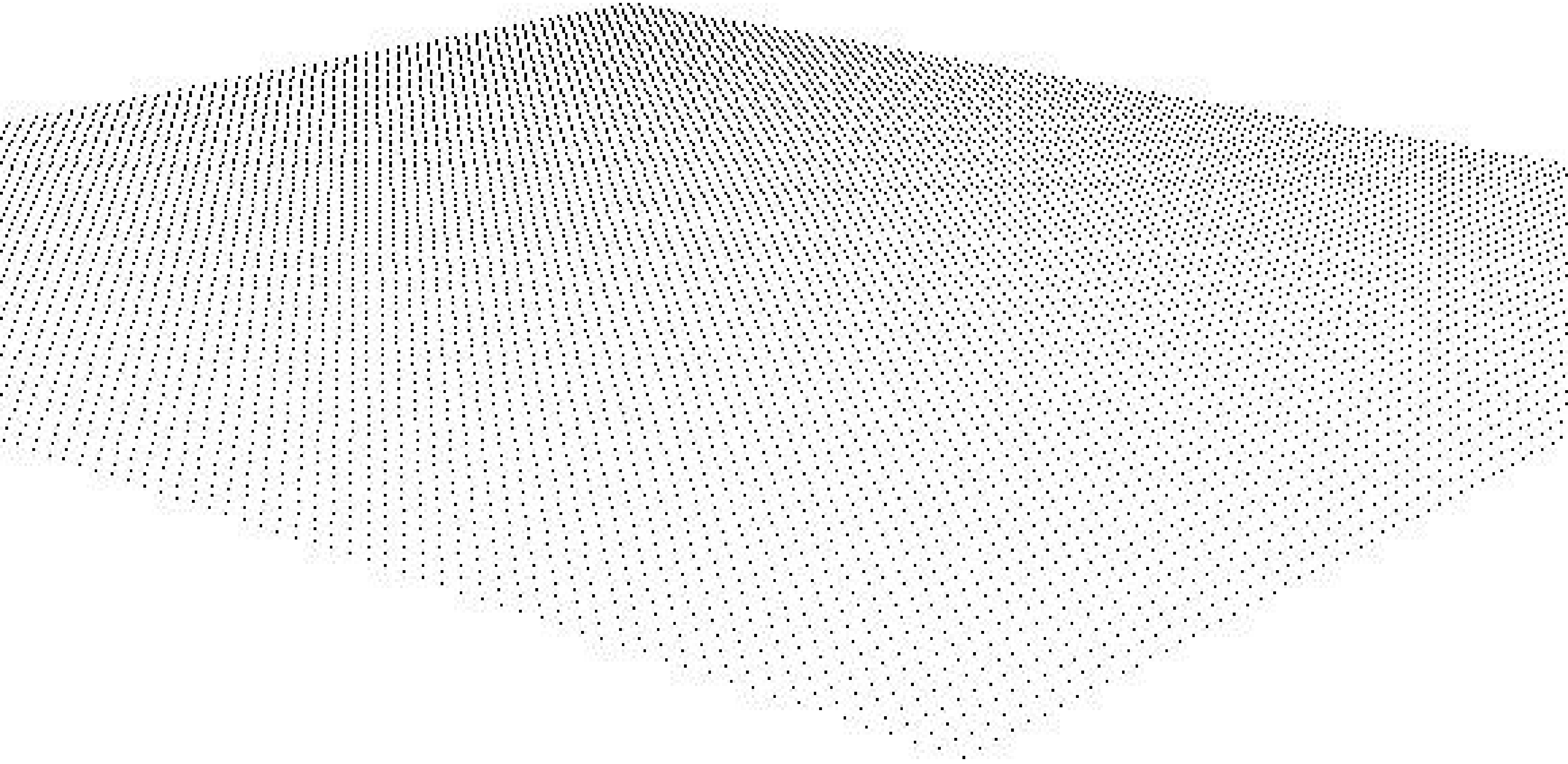
Ground Texture Antialiasing

- Ground texture drawn from vector outlines using analytically antialiased trapezoids
- Chooses ground resolution to match screen
- Achieves high-quality anisotropic antialiasing



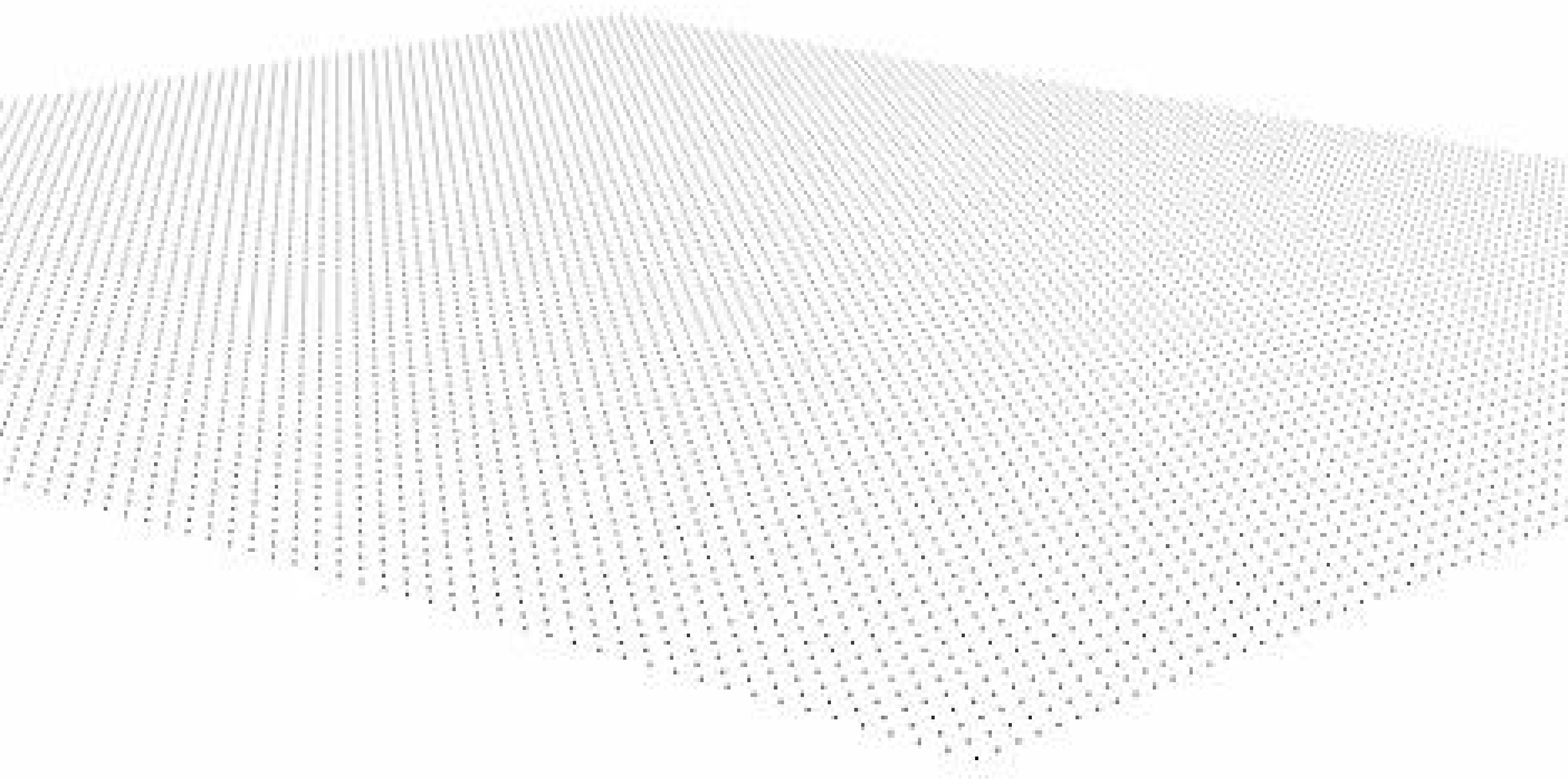
Splat Aliasing

- **Aliased splat geometry: lines break up and wobble**



Splat Antialiasing

- **Antialiased splats: lines stay smooth and clean**



Penumbra Limit Map for Soft Shadows

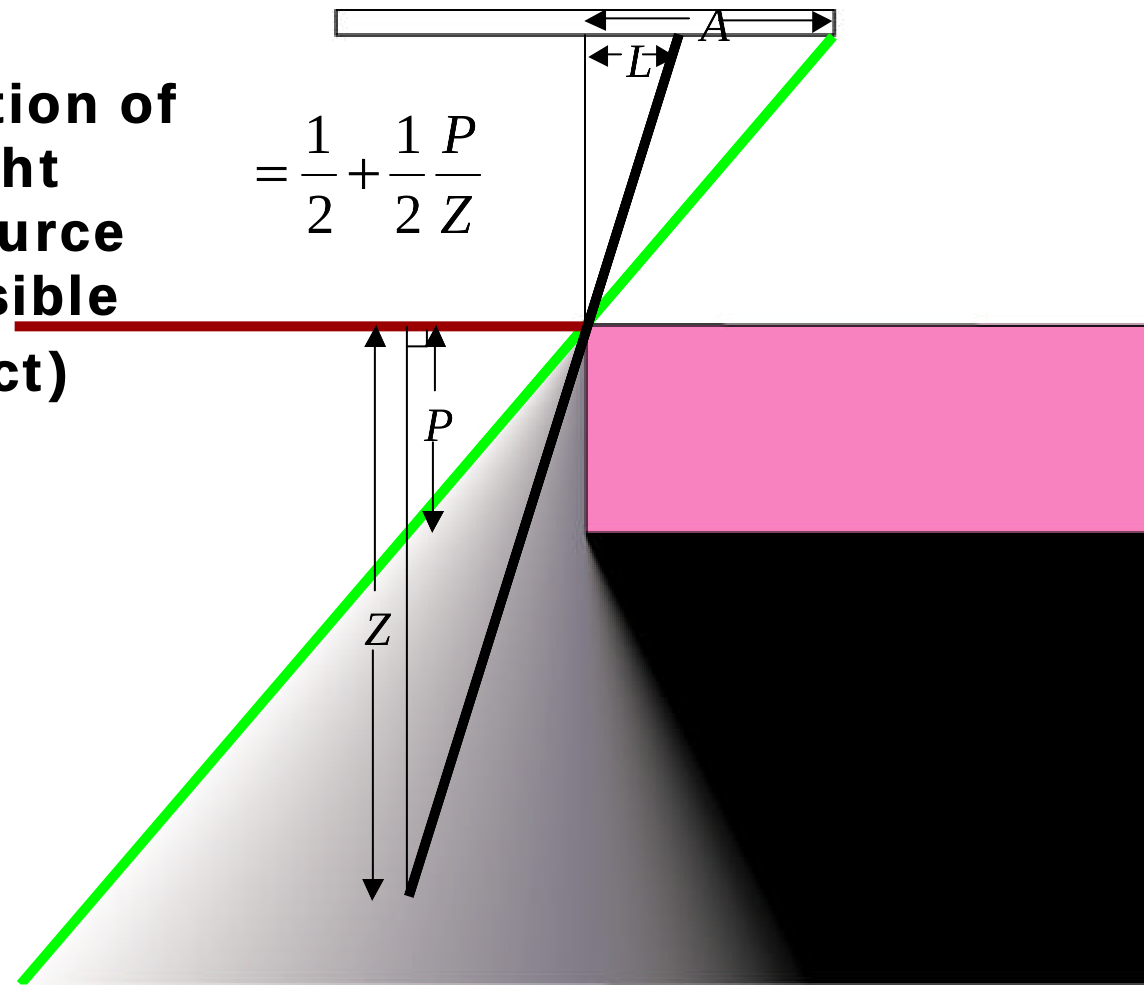
Quality: Soft Shadows

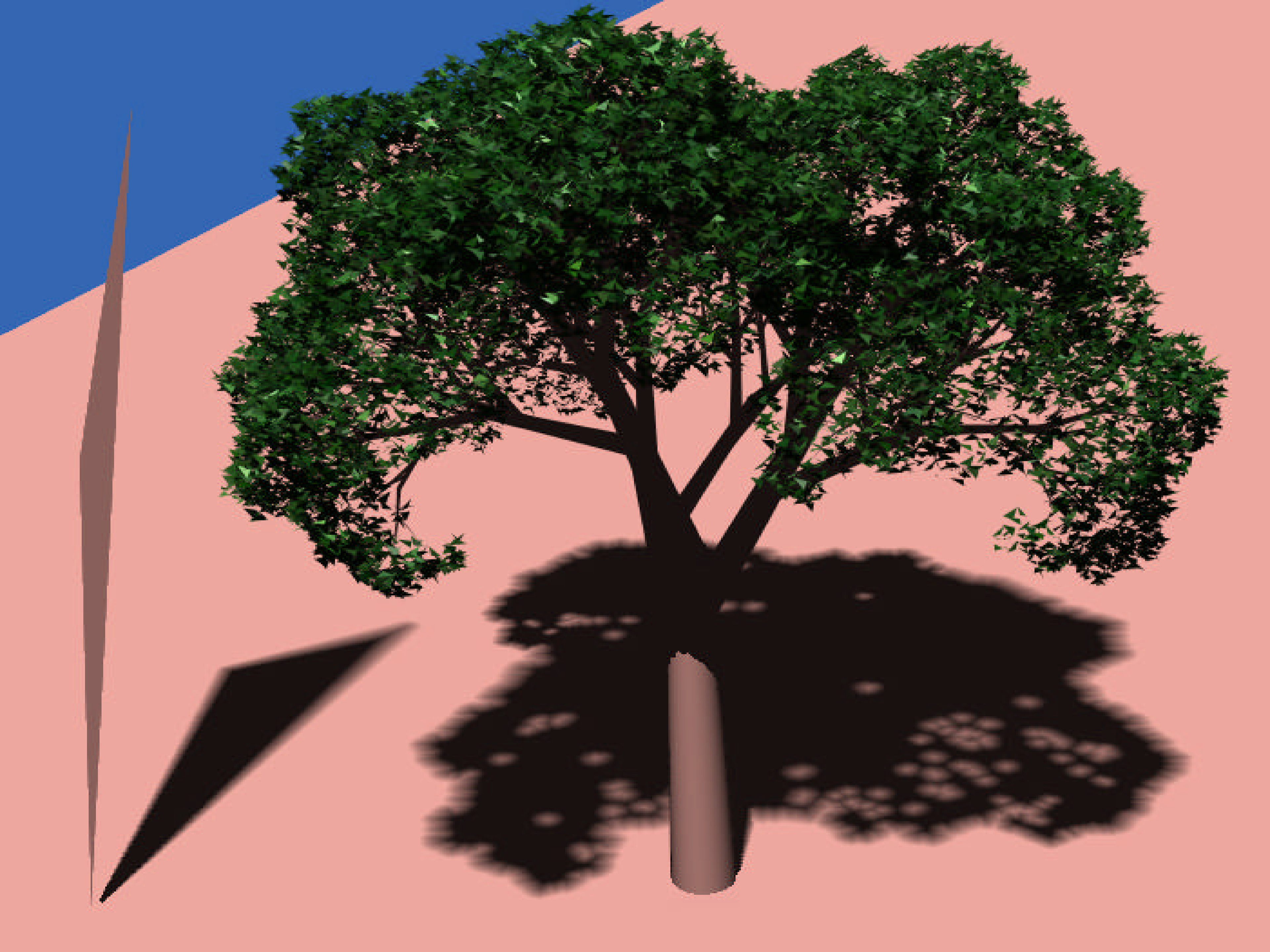
- Extended light sources cast fuzzy shadows
  - E.g., the sun
- Prior work
  - Ignore fuzziness
  - Point sample area source
  - New faster methods [Hasenfratz 03 survey]
- New method based on a discrete, easy-to-parallelize shadow map

Penumbra Limit Shadows

- **Main Contribution: new method physically correct**
- **New method very interpolation-friendly**
  - **Penumbra limit values (green) are planar**

Fraction of light source visible (exact) $= \frac{1}{2} + \frac{1}{2}\frac{P}{Z}$





Large Models

Scale: Kilometers

- **World is really big**
  - **Modeling it by hand is painful!**
- **But databases exist**
  - **USGS Elevation**
  - **GIS Maps**
  - **Aerial photos**
- **So *extract* detail from existing sources**
  - **Leverage existing manual labor**
- **Gives *reality*, which is useful**

Practical Difficulties

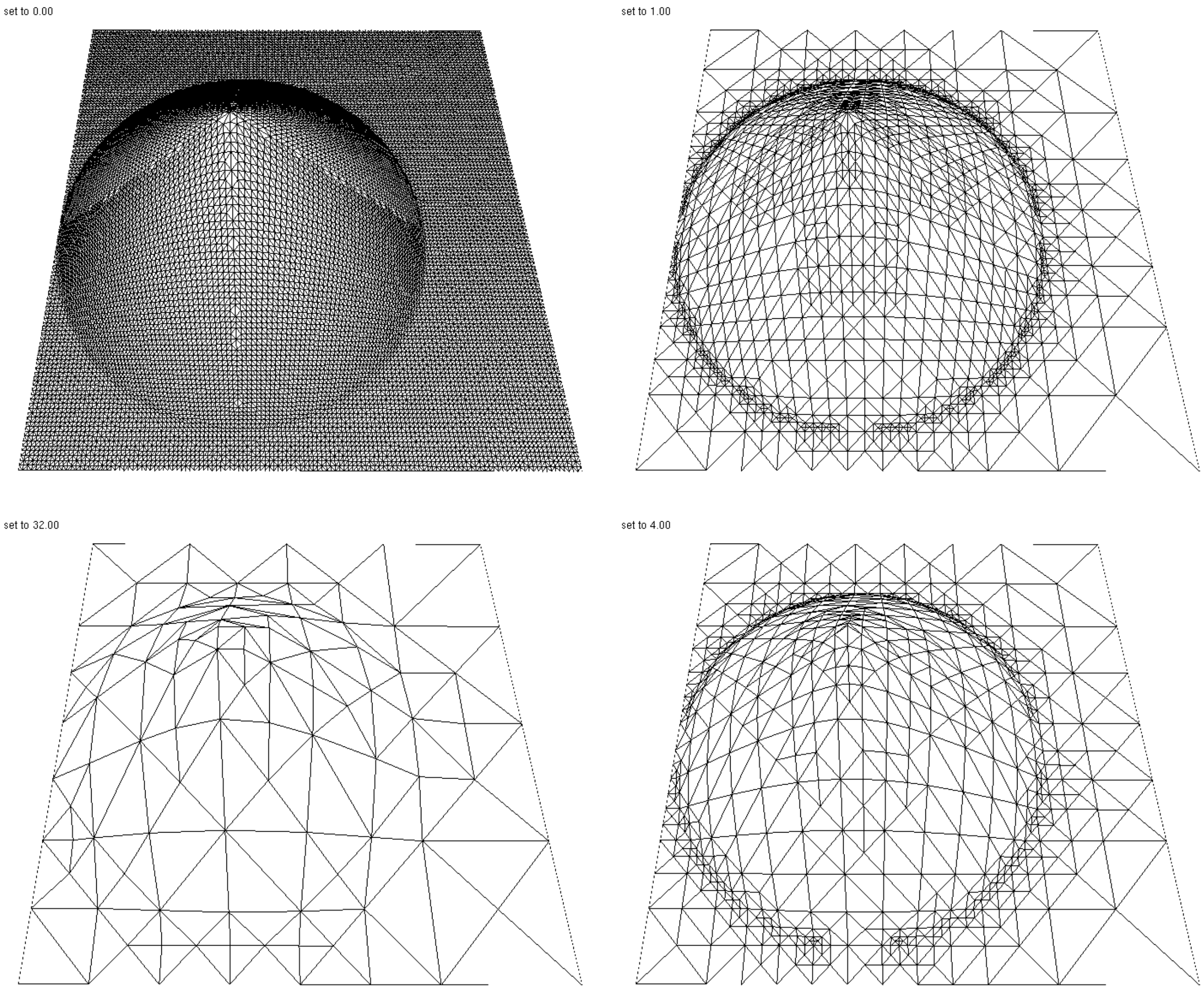
- Map projections
  - UTM, ILCS
  - Curvature of Earth
- Undocumented and bizarre formats
- Formats designed for 2D; need 3D
  - Extrusion
- Inconsistencies
  - 1997 vs 2004
- *Still* much easier than by hand…

Terrain Traversal

- **Cannot simply dump all terrain geometry into graphics card**
  - **Too many polygons**
- **Must simplify terrain geometry during traversal**
  - **But must preserve fidelity**
  - **View-dependent level of detail**
- **Standard method [Lindstrom 03]**
  - **With a few minor improvements**

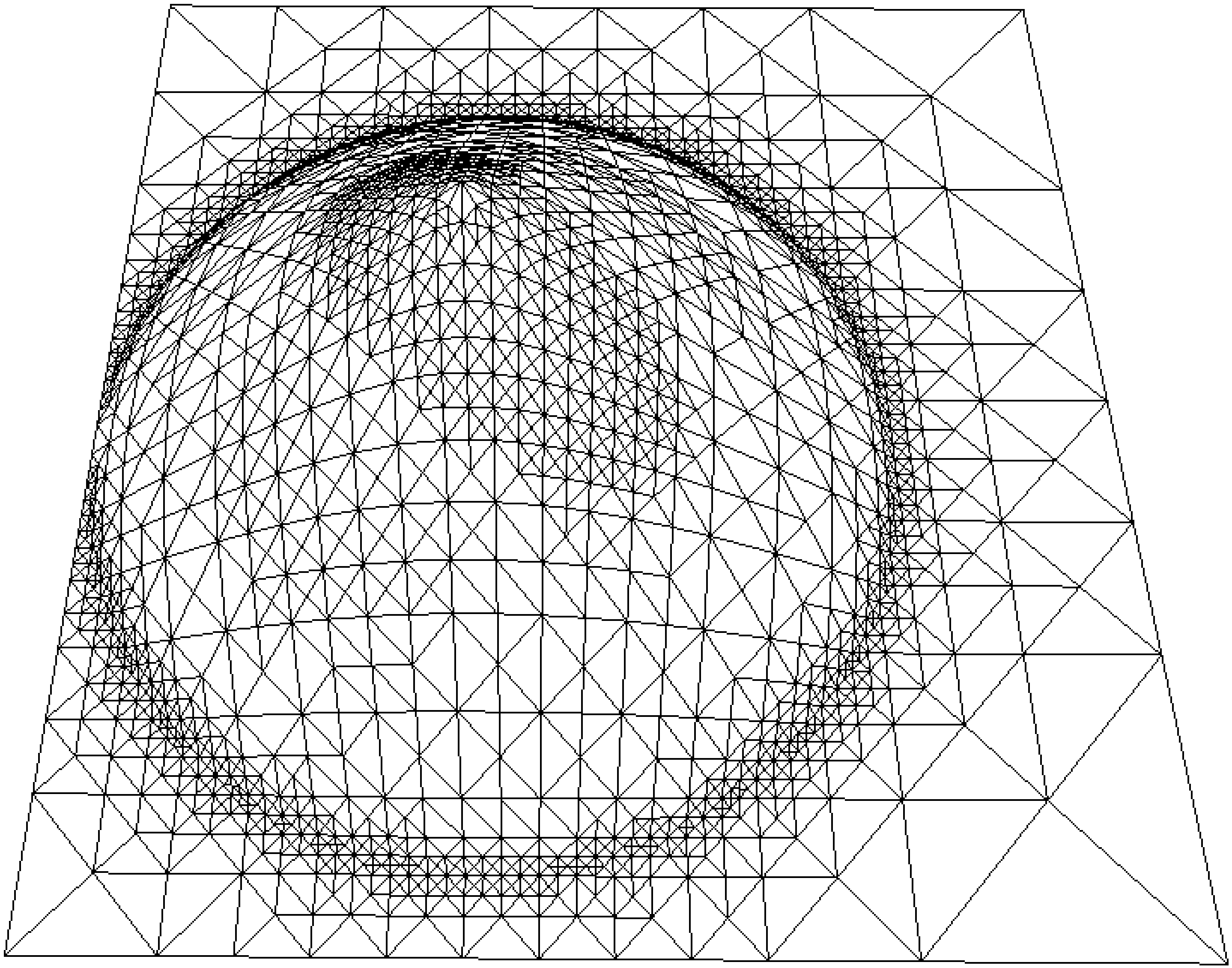
Terrain Decomposition

- **Terrain level-of-detail: expand until screen error drops below threshold**



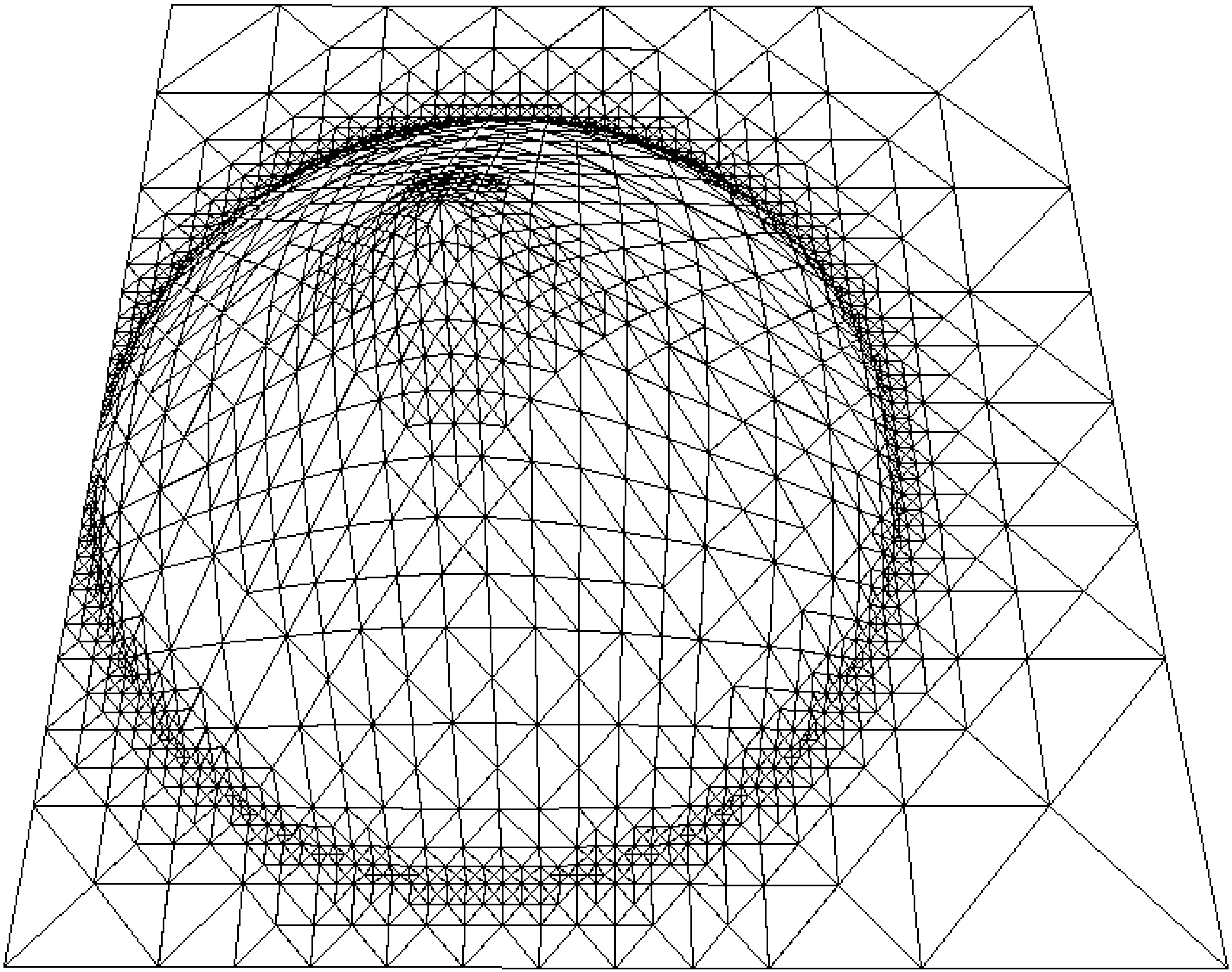
Terrain Decomposition

- **Lindstrom terrain: split quads at even/odd levels**



Terrain Decomposition

- **Optimized terrain: split quads along lower-error axis**

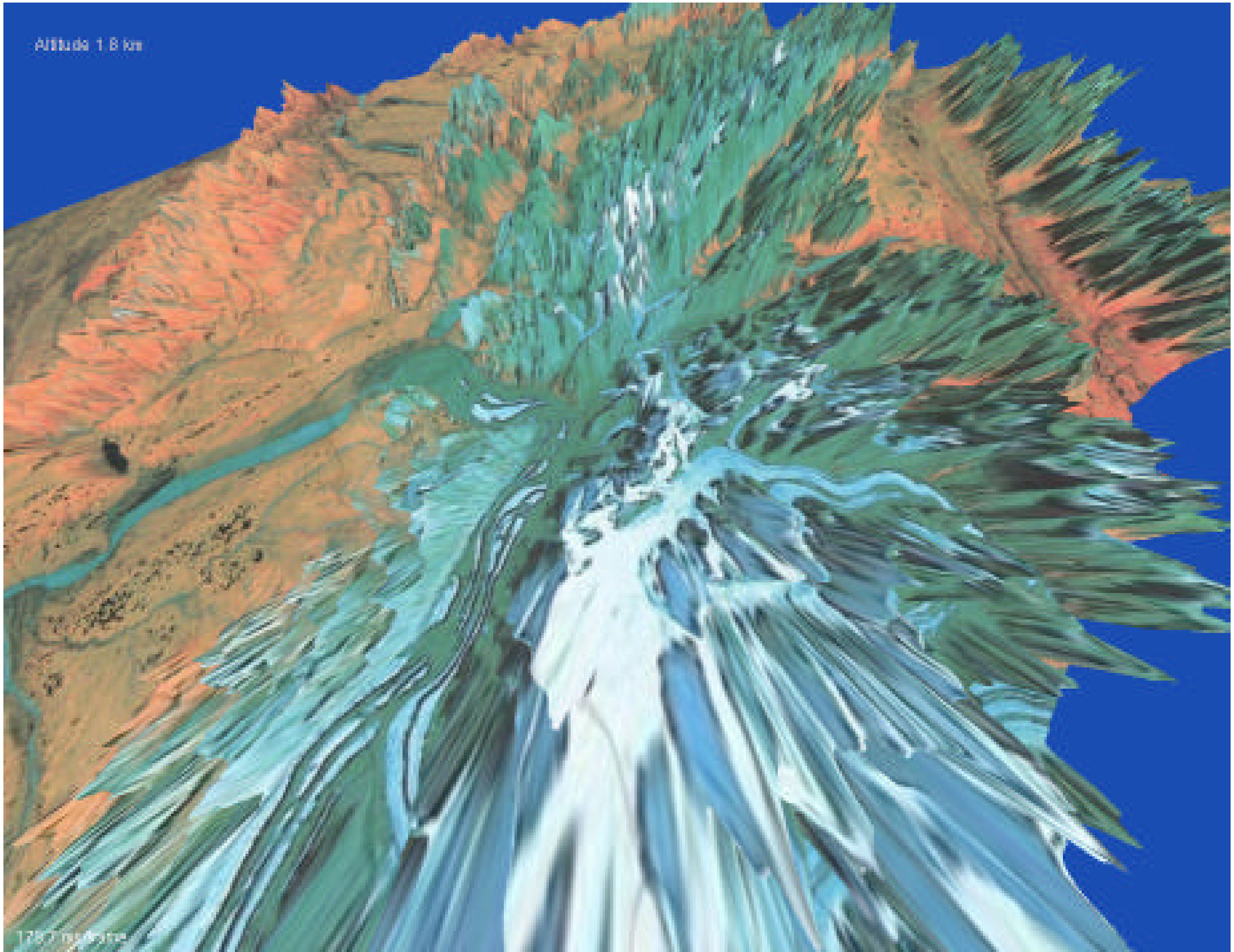


Terrain Painter's Algorithm

- **Conventional Z-buffer terrain can be extracted in arbitrary order**
- **But painter's algorithm requires strict back-to-front rendering**
  - **So recursively traverse terrain in back-to-front order**
  - **Expand children in back-to-front order**

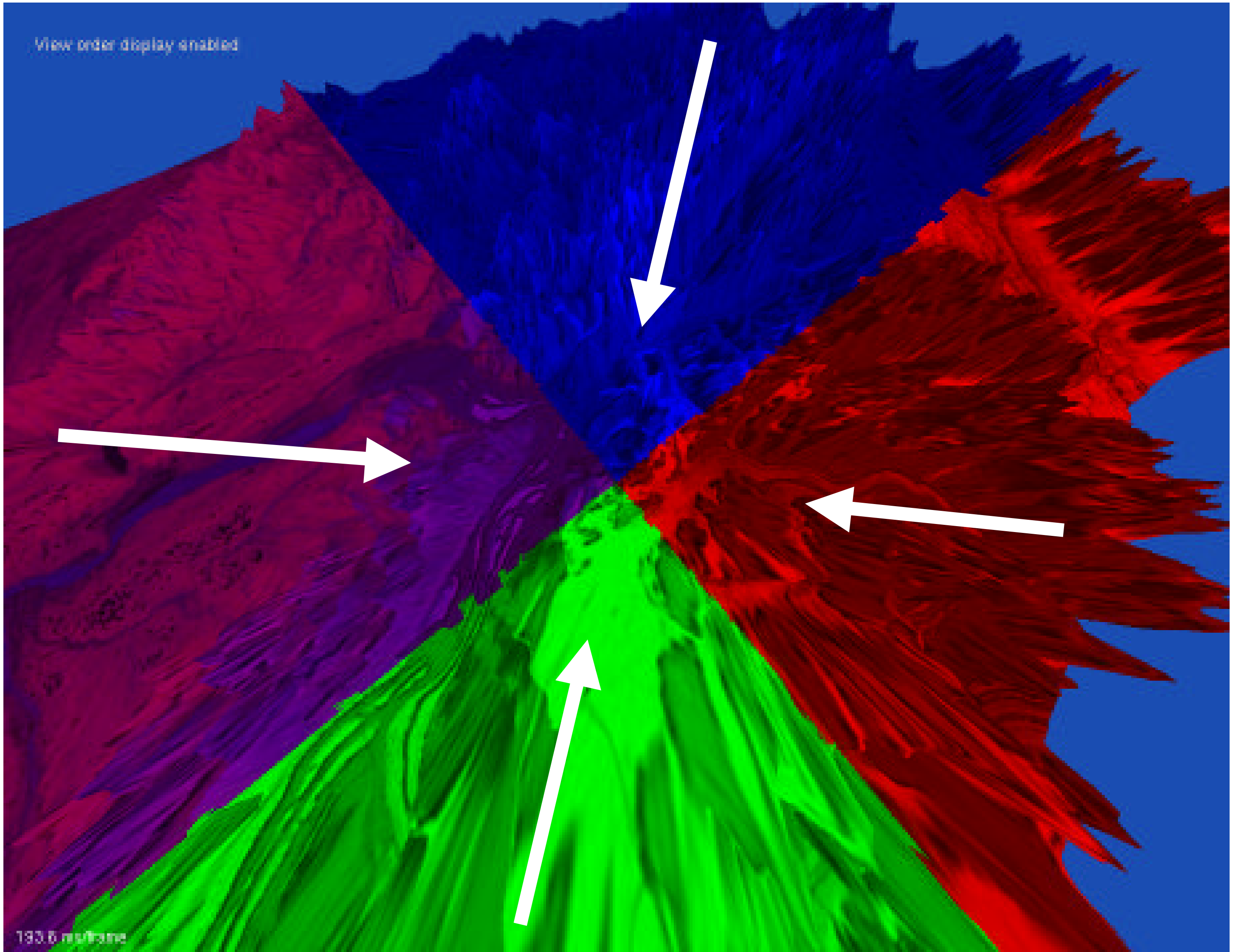
Terrain Painter's Algorithm

- **Extreme Wideangle shot of Denali Nat'l Park**



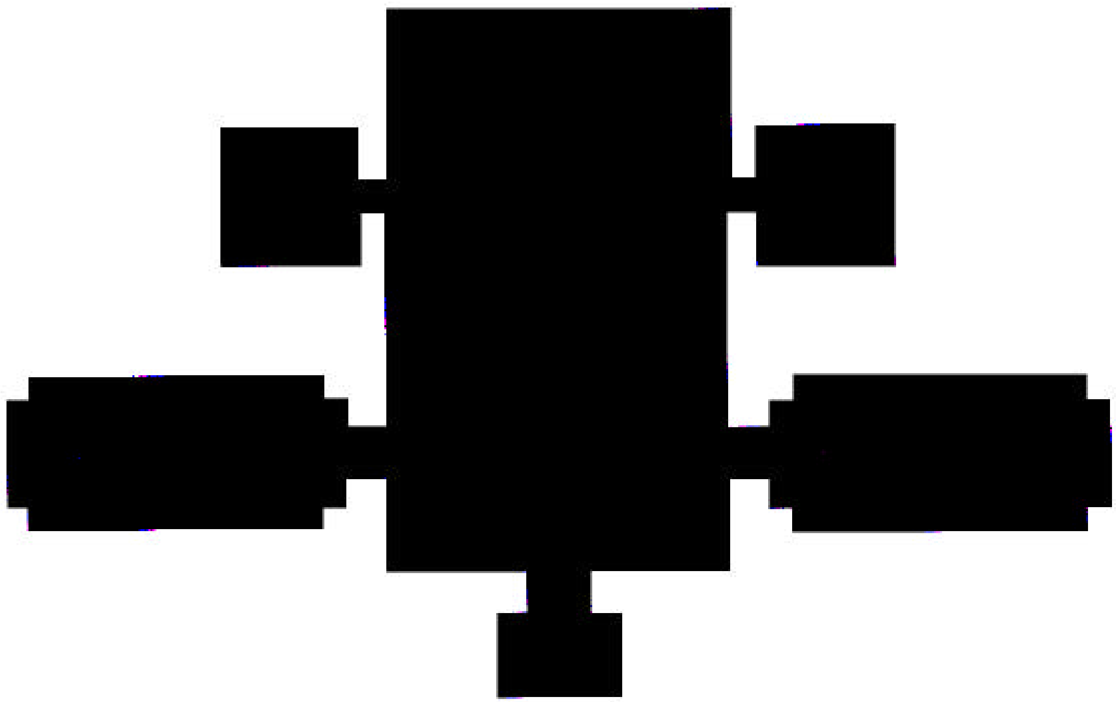
Terrain Painter's Algorithm

- **Colored by traversal order**



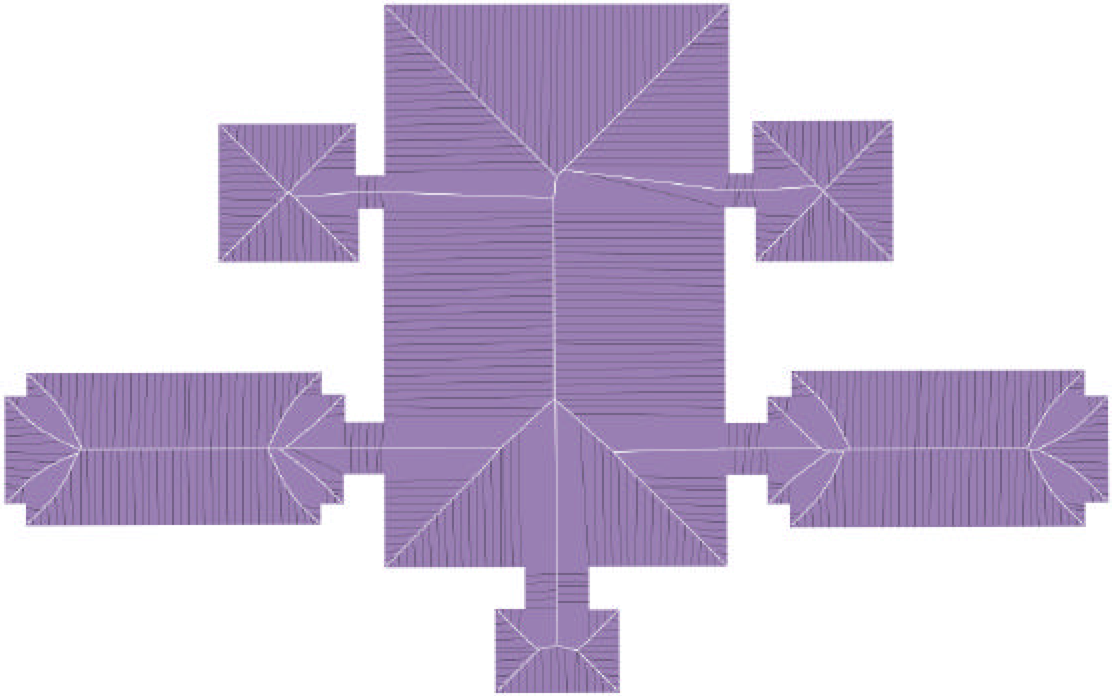
Roof Extrusion

- Only have building outlines, not details of roof topology or even height
- Must synthesize plausible roof shape for hundreds of buildings
- Building outlines contain lots of colinearity and other degeneracies!



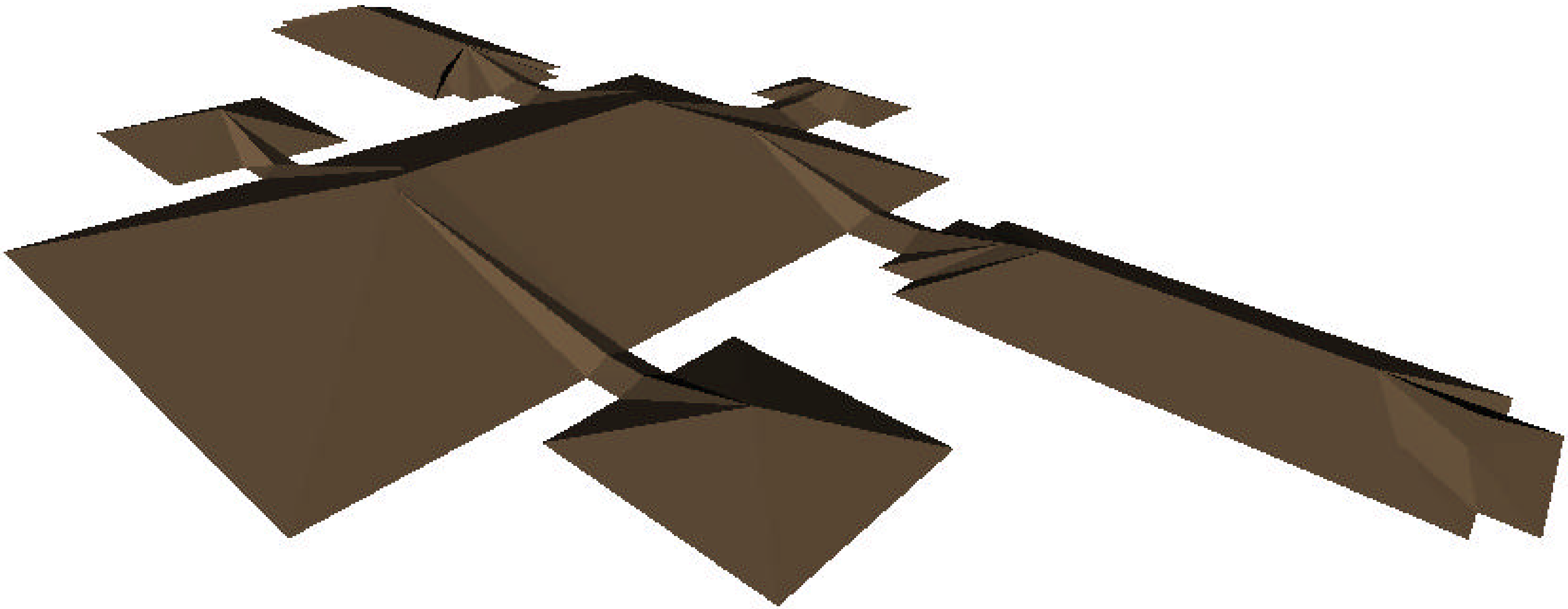
Roof Extrusion

- New (?) triangulation based on Voronoi diagram
  - Triangulates medial axis and outline
  - Plausible approximation of real roofs
- Medial axis approximately follows ridgeline
- Special “cell edges” run downslope, can highlight to draw water channels



Roof Extrusion

- Procedure is fast and robust
  - Built on Fortune's sweepline algorithm
- Works for all campus buildings without problems
- Simplify resulting roof mesh using quadric simplification [Garland 97]



Contributions and Conclusions

Contributions: Parallel Computing

- **Charm++ Array Manager**
  - **Parallel migratable objects support**
    - **Scalable Creation, deletion, messaging, migration**
    - **Used here to represent chunk of geometry for impostor rendering**
  - **Collectives with migration [Lawlor 03]**
    - **Used here to distribute new viewpoints to impostors**
- **Charm++ PUP Framework**
  - **Introspection for C++ objects**
  - **Complex cross-platform communication protocols made easy [Jyothi and Lawlor 04]**
  - **Used here for impostors:**
    - **To/from disk files (scene I/O)**
    - **To client from server**
    - **Between processors of parallel machine for load balance**
- **CCS Protocol**
  - **Fast, portable network connection to parallel machines [Jyothi and Lawlor 04]**
  - **Works even with both ends behind firewalls or NAT**
  - **Used here to connect parallel impostor server to client**

Contributions: Parallel Rendering

- **Parallel Impostors technique for**
  - **Additional rendering power**
    - **More geometry per frame**
    - **Better rendering algorithms**
    - **Quality antialiasing**
  - **Improved bandwidth usage**
    - **Impostor reuse cuts required bandwidth**
  - **Increased latency tolerance**
    - **Client can always draw next frame using existing impostors**
    - **No dropped frames from network glitches**

Contributions: Quality Rendering

- Techniques for
  - Antialiased geometry
    - Analytic filtering and smooth splats
  - Quality lighting
    - Soft shadows via Penumbra Limit Maps
    - Global illumination via Impostor GI
  - Large worlds
    - GIS and Terrain tweaks
  - Procedural geometry generation
    - IFS Bounding [Lawlor and Hart 03]
- Cost of these techniques is <u>affordable</u> with Parallel Impostors

Total Lines of Code

- Conservative total of <u>63K</u> lines of C++ code (with some C)

- Parallel-Rendering specific: <u>16K</u> lines
  - 9K Rendering and IFS support (for campus model)
  - 3K LiveViz3d server library (parallel impostors)
  - 1K LiveViz2d server library (screen shipping)
  - 1K Campus server code
  - 1K Campus client library
  - 1K Campus building assembly

- Graphics Infrastructure: <u>31K</u> lines
  - 10K 2D antialiased rendering library
  - 8K Matrix, vector, and other math
  - 6K PostScript interpreter
  - 3K Terrain system
  - 3K Geospatial / map libraries
  - 1K Raytracer library

- Parallel Infrastructure: <u>16K+</u> lines (CVS: <u>47K</u>)
  - 4K Array Manager
  - 4K Common data structures
  - 3K PUP Framework
  - 2.5K CCS Protocol

- Unrelated UIUC code: 25K lines
  - 7K FEM Framework
  - 4K CSAR Remeshing
  - 3K NetFEM client and server
  - 3K Data transfer library
  - 2.5K Collision library
  - 2K Multiblock framework
  - 1.5K TCharm library
  - 1.5K CSAR Makeflo

Future Work

- Camera motion prediction
  - Impostor prefetching
- Multi-impostor interpolation
  - Lightfield-style direction capture
- Fully hierarchical traversal
  - Split down to leaf and branch
- Integration with Impostor Global Illumination

http://charm.cs.uiuc.edu/users/olawlor/academic/thesis/

Backup Slides

Demo

Campus with Pure OpenGL

Campus with Parallel Impostors

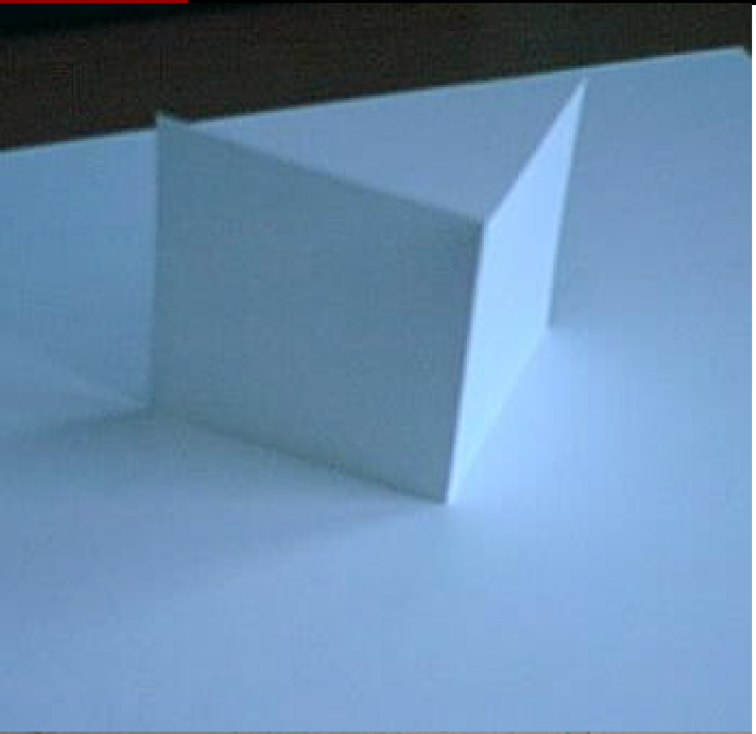
Impostor Frames

Importance of Computer Graphics

- "The purpose of computing is insight, not numbers!" R. Hamming
- Vision is a key tool for analyzing and understanding the world
- Your eyes are your brain's highest bandwidth input device
  - Vision: >300MB/s
    - 1600x1200 24-bit 60Hz
  - Sound: <1 MB/s
    - 96KHz 24-bit stereo
  - Touch: <100 per second
  - Smell/taste: <10 per second
- Plus, it looks really cool...

Impostor Global Illumination

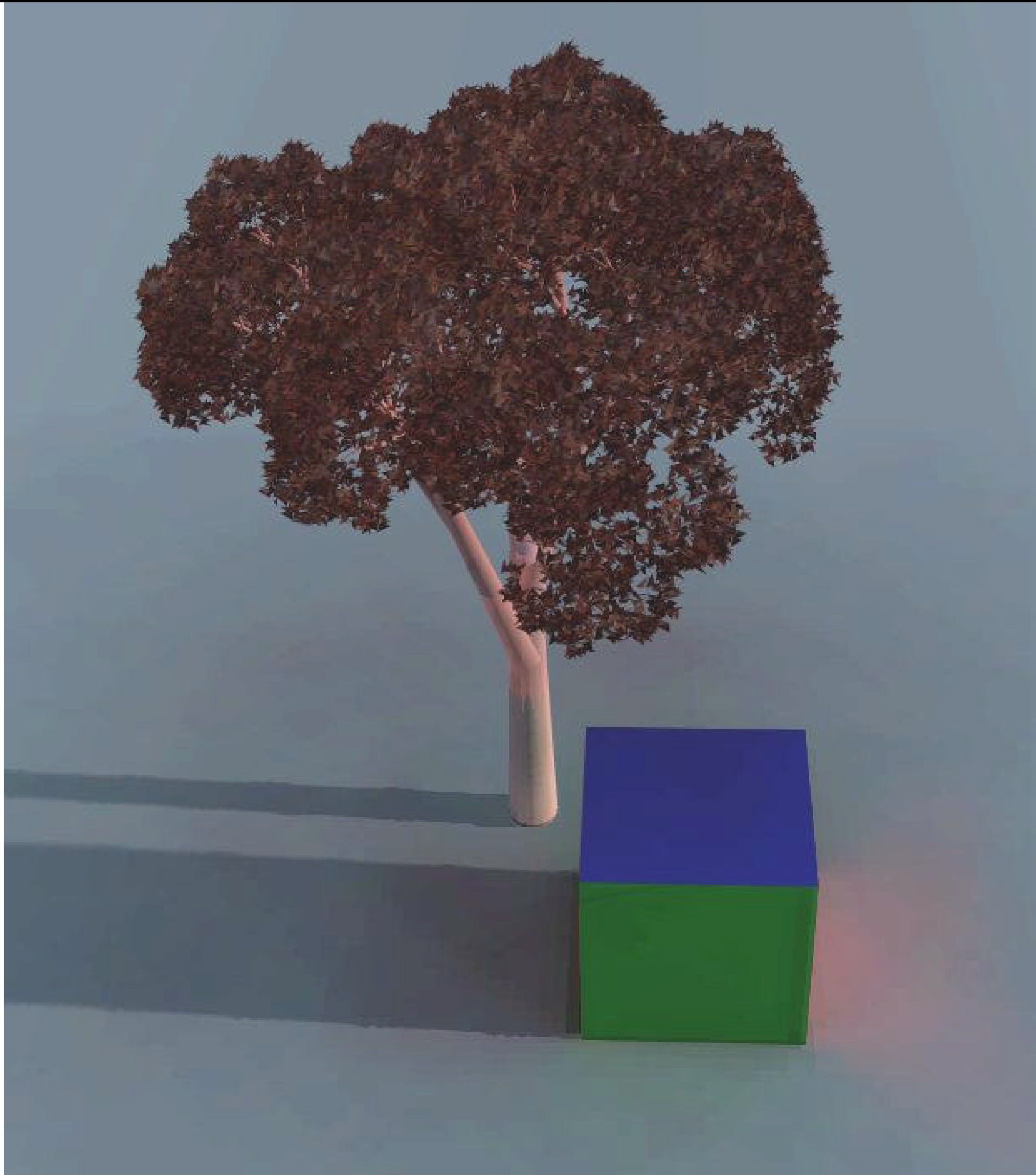
Quality: Global Illumination



- **Light bounces between objects (color bleeding)**
  - ***Everything* is a distributed light source!**
- **Prior work**
  - **Ignore extra light**
    - **“Flat” look**
  - **Radiosity**
  - **Photon Mapping**
  - **Irradiance volume [Greger 98]**
  - **Spherical harmonic transfer functions**

Impostor Global Illumination

- Sweep plane through scene, accumulating light from objects
- Identical to standard voxel/cubemap parameterization, but much faster to compute
- Allows geometry to be filtered during sweep

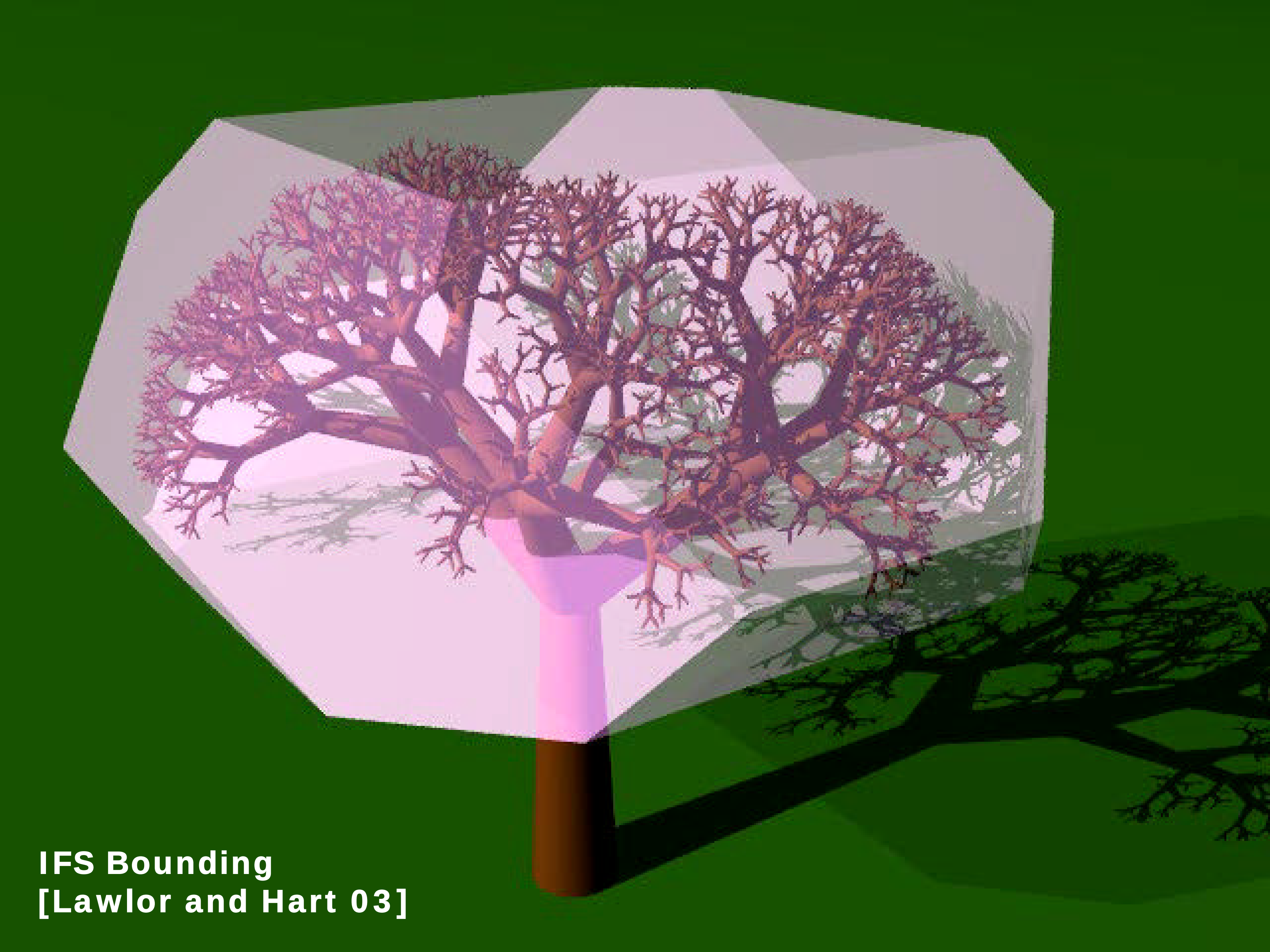


Complex Geometry

Detail: Complicated Geometry

- World's shape is complicated
- But lots of repetition
- So use subroutines to capture repetition

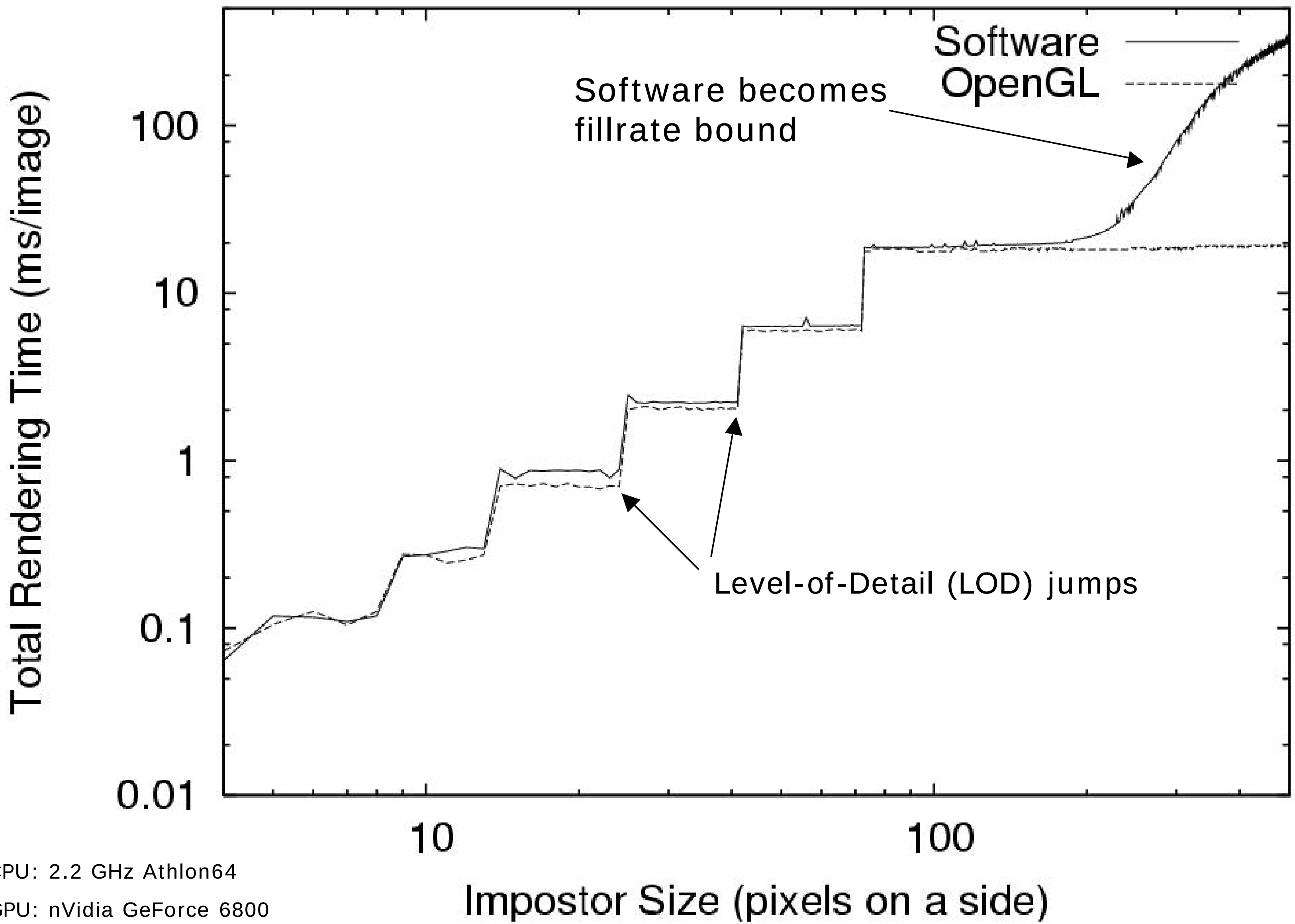
[Prusinkiewicz, Hart]



IFS Bounding
[Lawlor and Hart 03]

Software vs. Hardware Rendering Rate

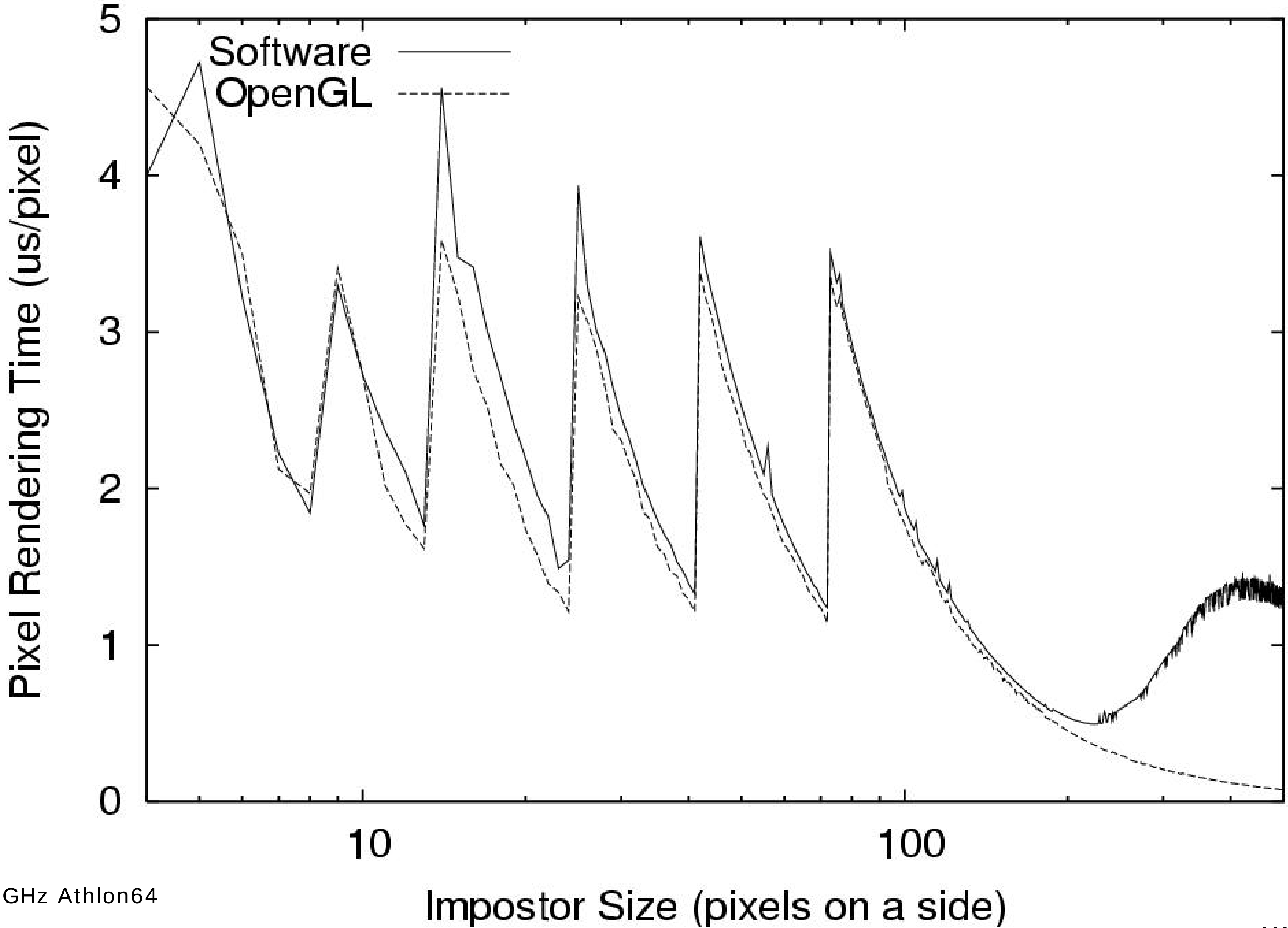
Rendering Time for Tree



CPU: 2.2 GHz Athlon64

GPU: nVidia GeForce 6800

Rendering Time per Pixel for Tree



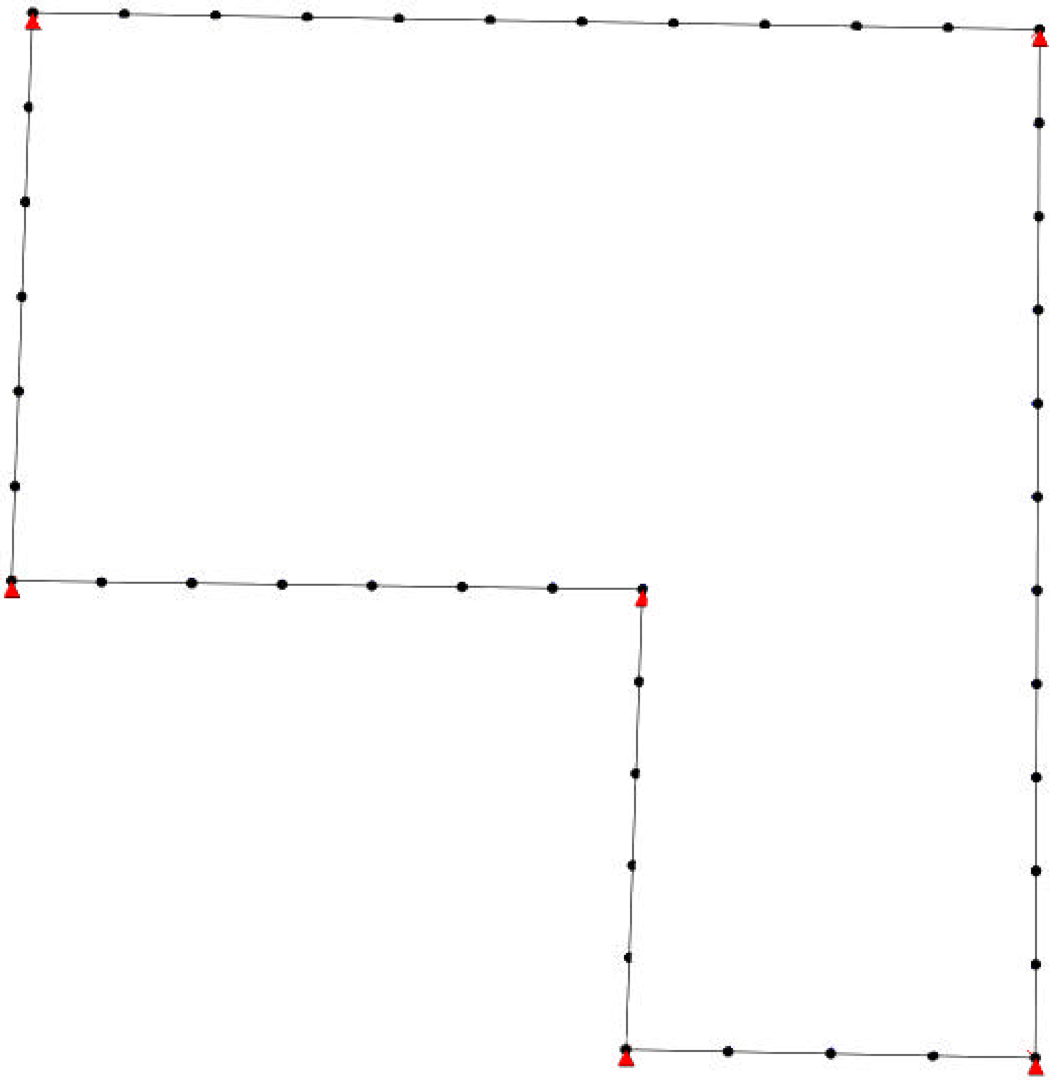
CPU: 2.2 GHz Athlon64

GPU: nVidia GeForce 6800

Roof Extrusion Details

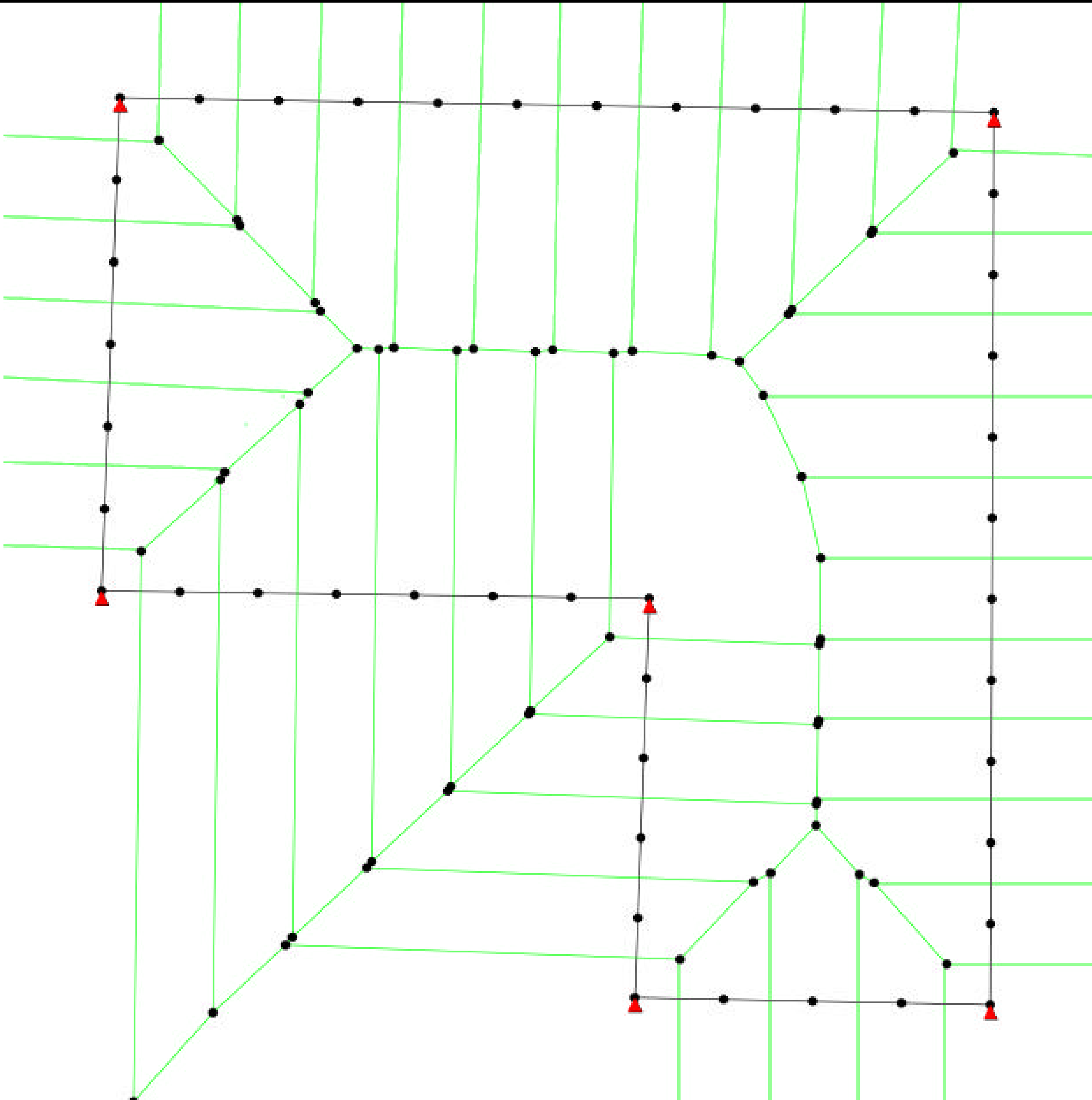
Roof Extrusion Steps

- Start with building outline
- Discretize outline into small pieces (20cm)



Roof Extrusion Steps

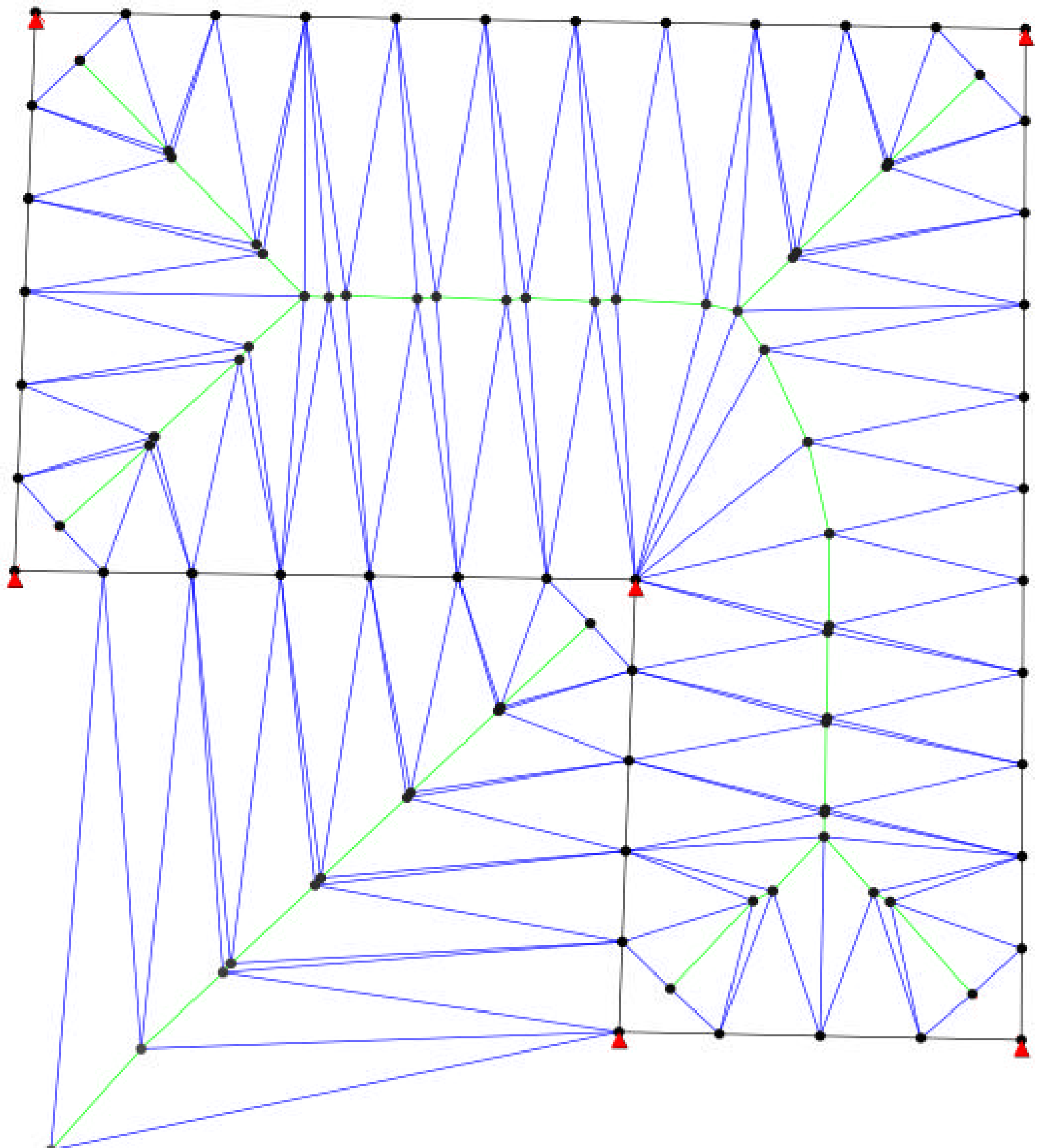
- **Compute Voronoi diagram of discretized outline**
- **Keep Voronoi vertices (center) and edges (green)**
- **Voronoi diagram approximates medial axis of building**



Roof Extrusion Steps

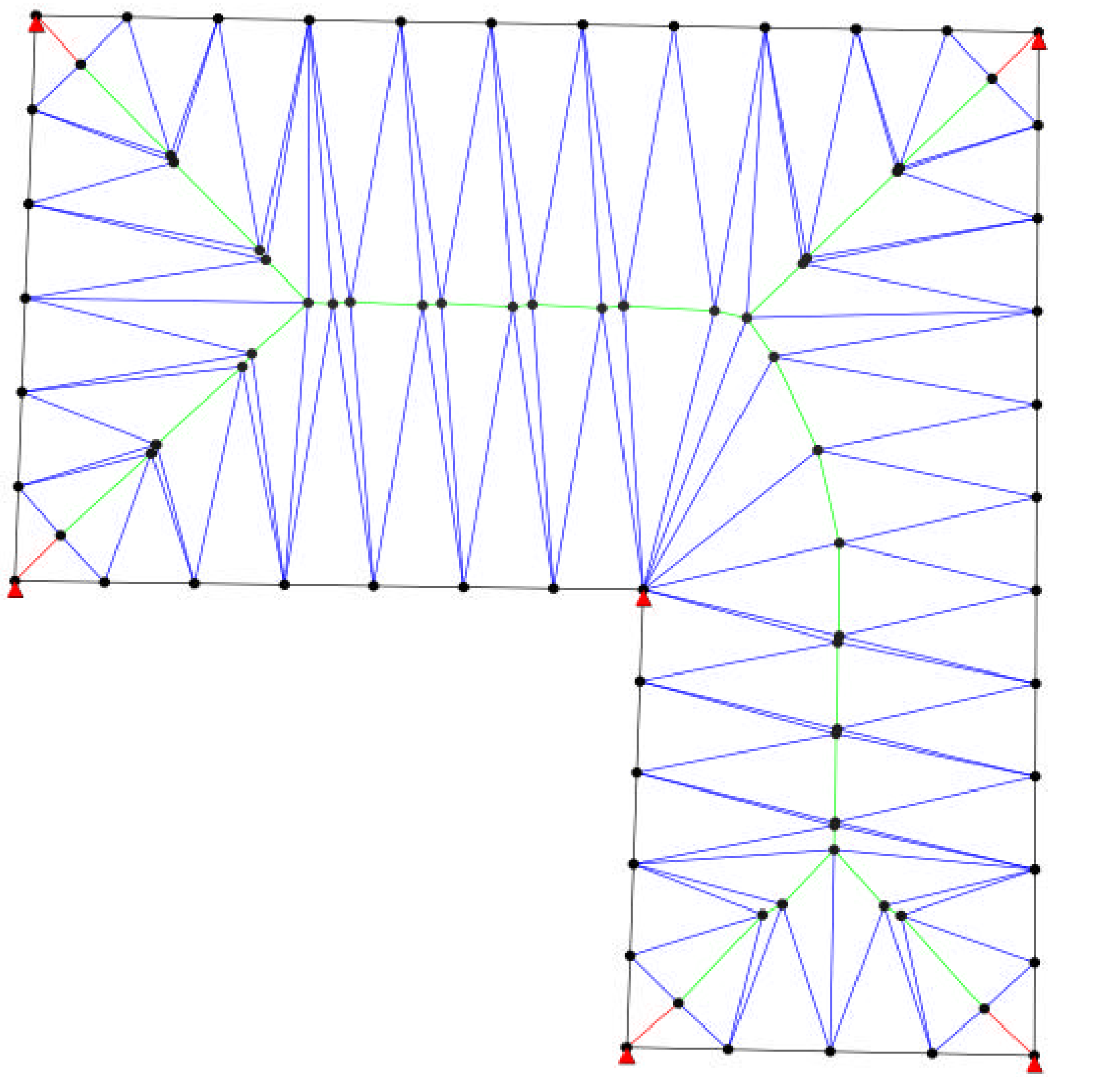
- For Voronoi edges that cross the old outline, delete the edge and connect the corresponding Voronoi vertices to their controlling set points using new edges (blue)

- The new edges cannot cross, because Voronoi cells are convex



Roof Extrusion Steps

- Remove Voronoi vertices that go outside the set
- Add Voronoi edges (red) to corner vertices (needed for acute corners)
- Result is a triangulation of the roof outline and medial axis
- Can now extrude to 3D and simplify



Roof Extrusion

- Procedure is fast and robust
- Worked for all campus buildings without problems