Lecture 13:
Reyes Architecture and Implementation

Kayvon Fatahalian
CMU 15-869: Graphics and Imaging Architectures (Fall 2011)

A gallery of images rendered using Reyes

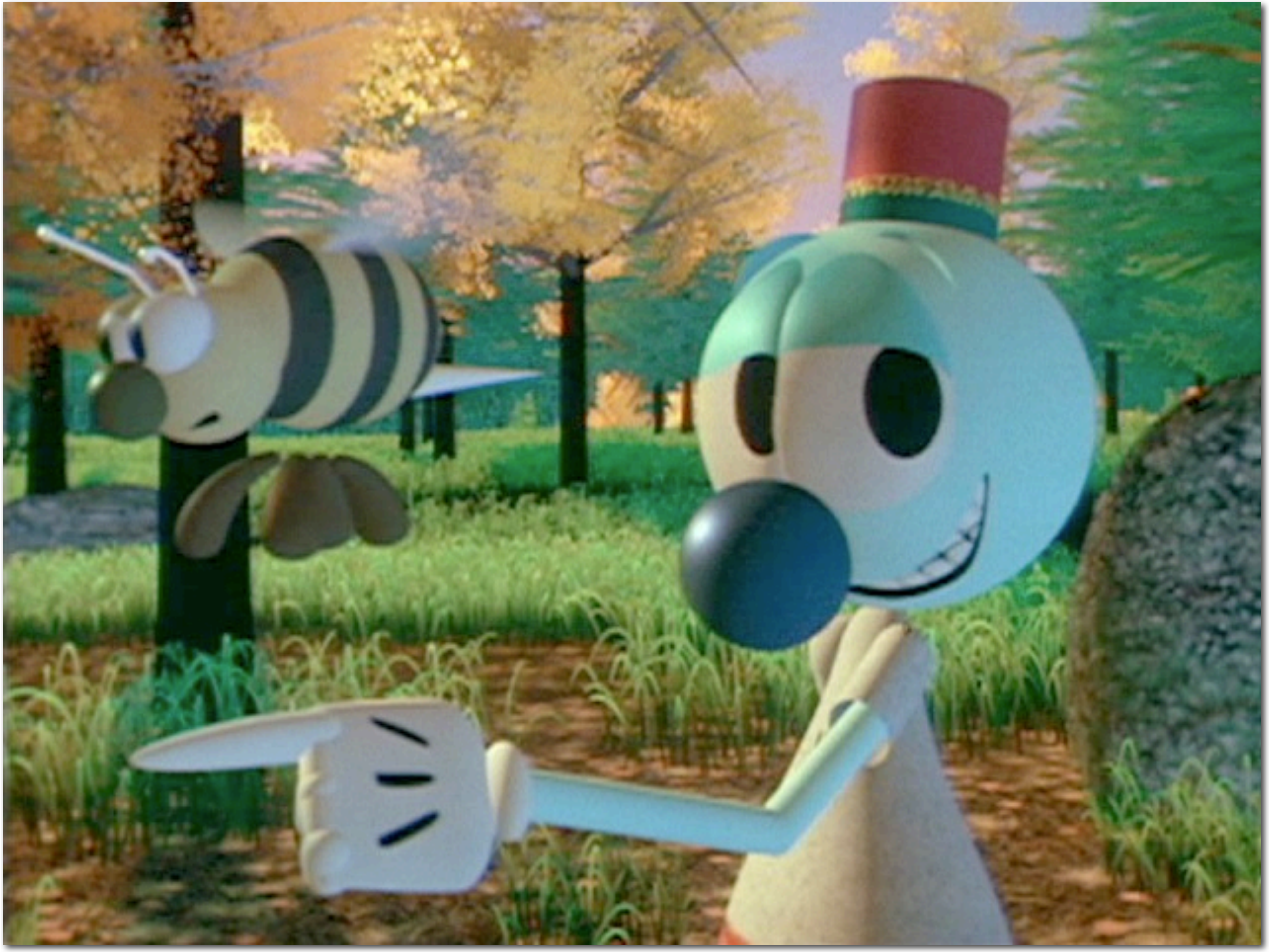


Image credit: Lucasfilm (Adventures of Andre and Walle B, 1984)

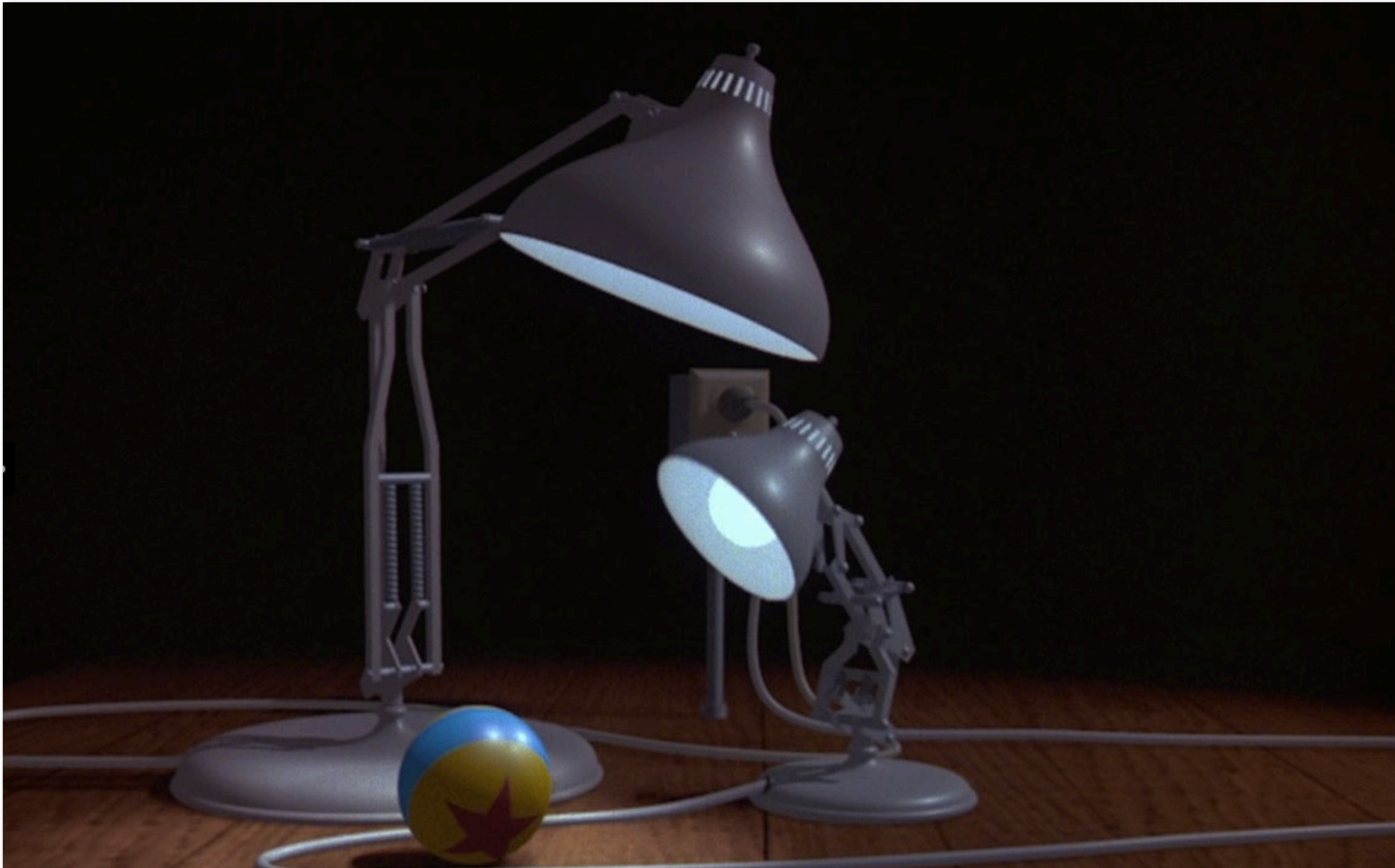


Image credit: Pixar (Luxo Jr., 1986)

Image credit: Pixar (Toy Story 2, 1999)

Image credit: Pixar (Wall-E, 2008)


Image credit: Pixar (Ratatouille, 2007)


Image credit: Pixar (UP, 2009)

Image credit: Pixar (UP, 2009)


Image credit: Pixar (UP, 2009)

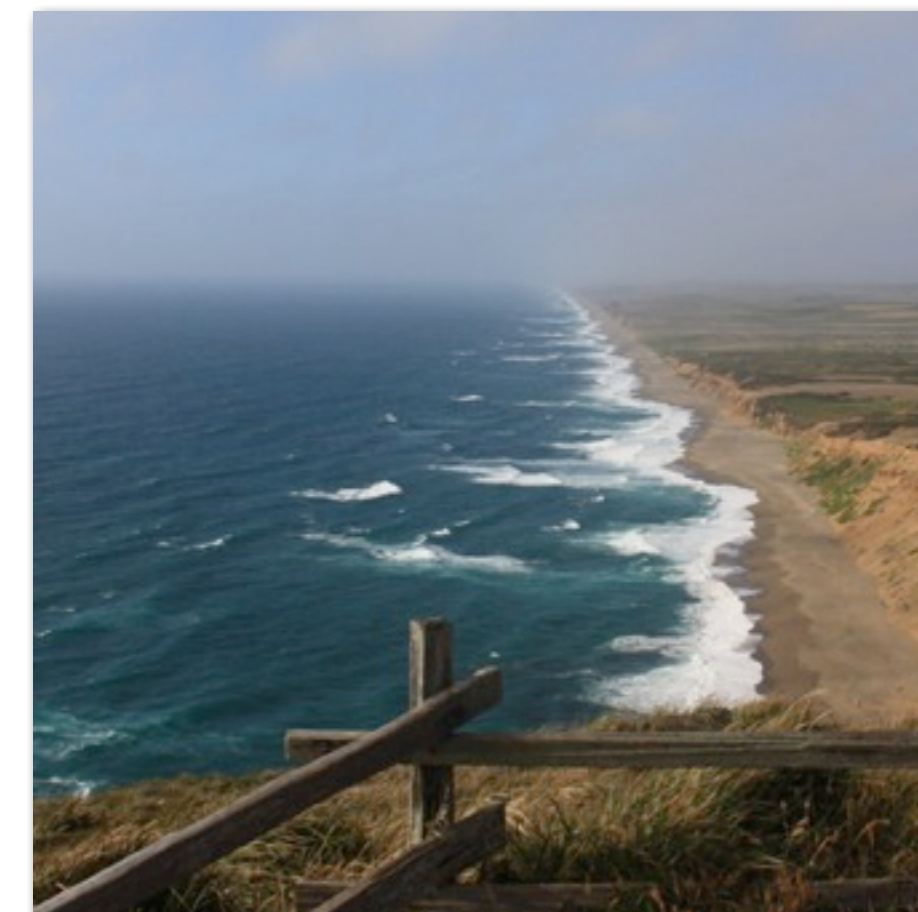
Image credit: Pixar (UP, 2009)

Image credit: Pixar (UP, 2009)

Image credit: Pixar (Toy Story 3, 2010)

The Reyes image rendering architecture

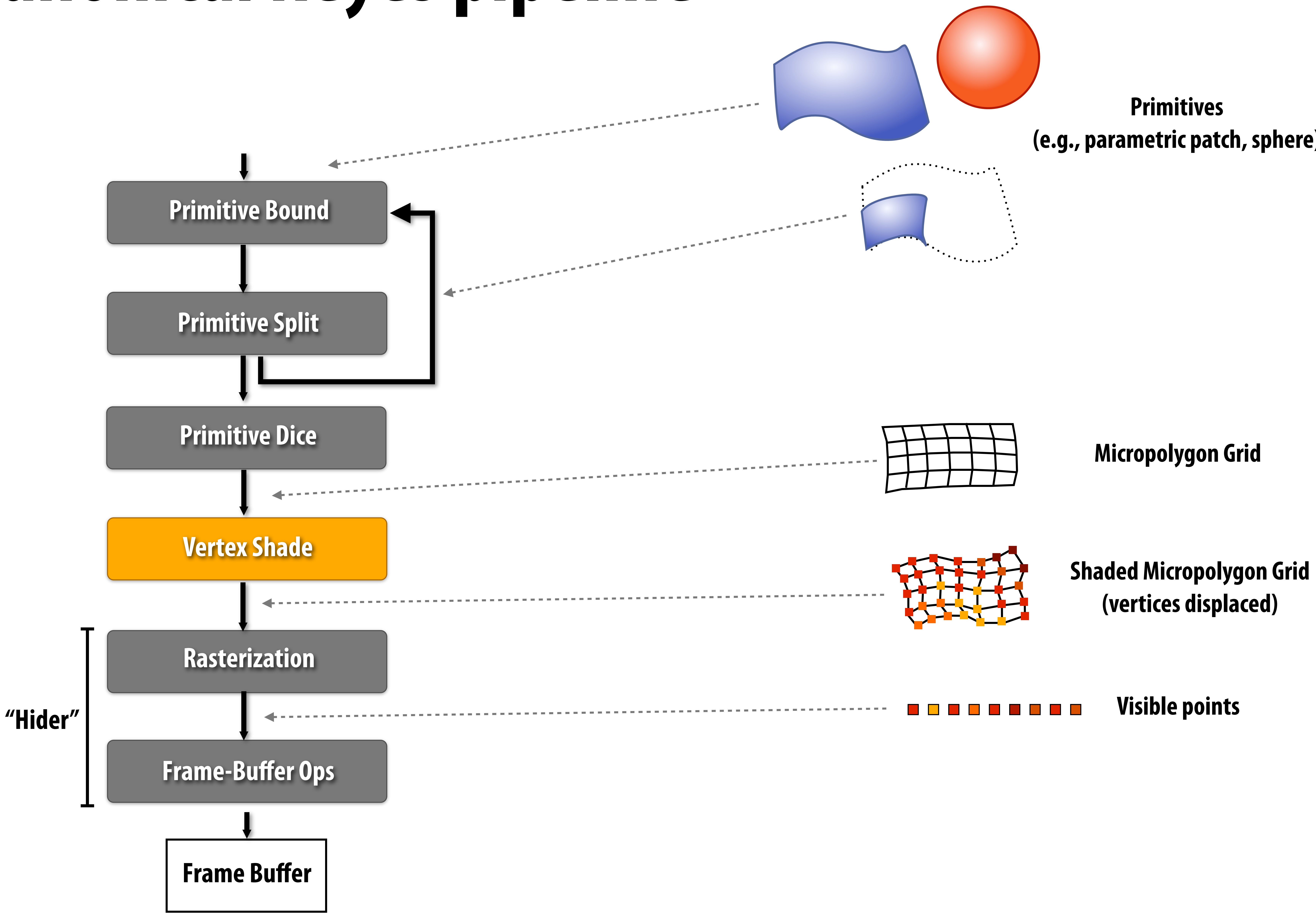
- Reyes: acronym for Renders Everything You Ever Saw
  - Also reference to Pt. Reyes, CA (just north of San Francisco)
  - Disagreement in graphics community about whether it is written Reyes or REYES. (Rob Cook says it's "Reyes")
- Developed at Lucasfilm (graphics group later became Pixar)
- Pixar's implementation is called Photorealistic Renderman (prman)
  - Renderman name was a take off on Sony Discman
- Rendering system for every Pixar film
  - And vast majority of film special effects



Reyes goals

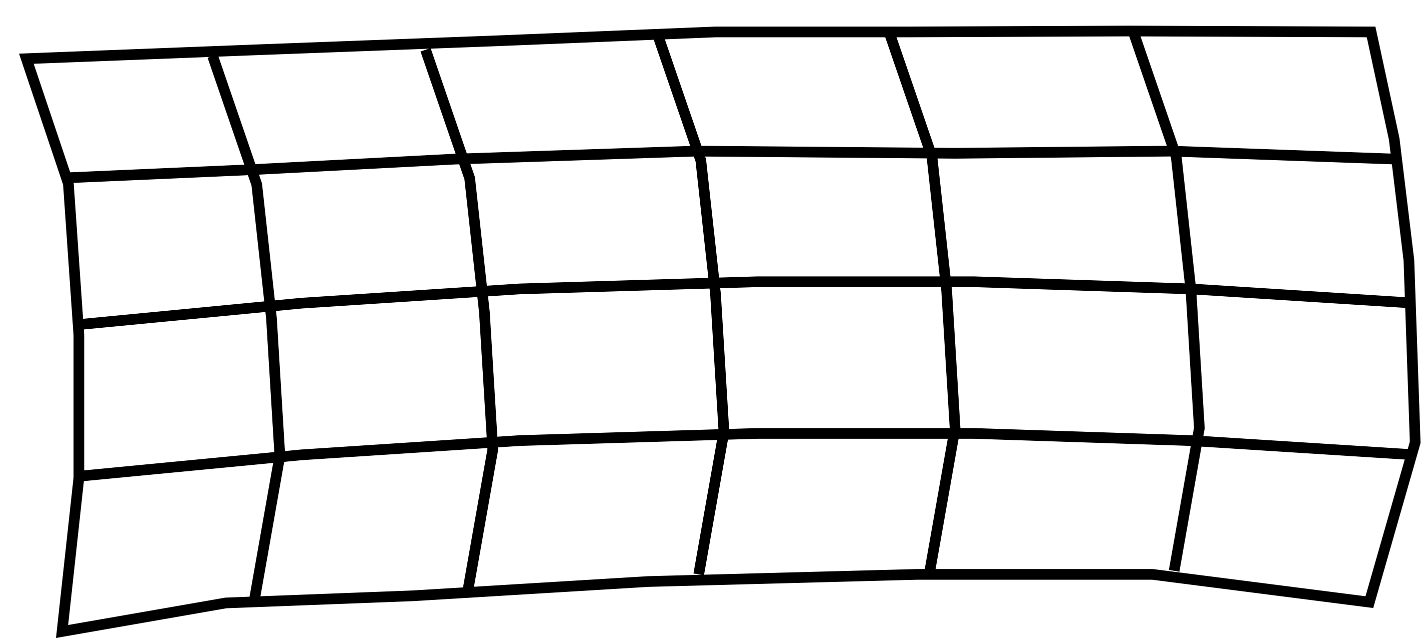
- High image quality: no faceting, no visible aliasing
- Handle massive scene complexity
- Support large diversity in models, shading, etc.
- High performance: achieve all of the above in “reasonable” rendering time (minutes/hours frame)

Canonical Reyes pipeline

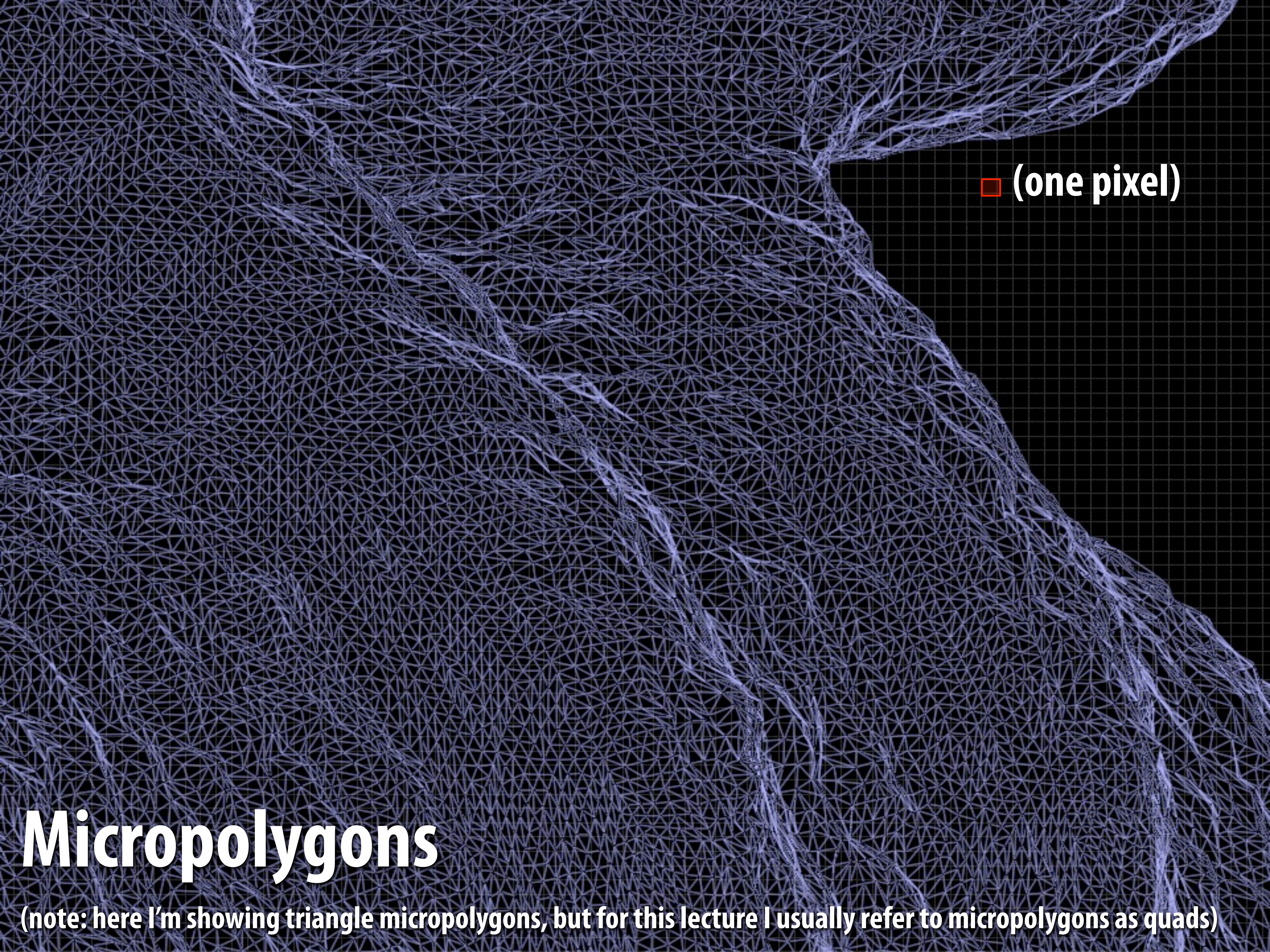


Definitions

- **Micropolygon = canonical intermediate representation in the Reyes pipeline. Expectation is that projected area <= 1 pixel**
- **Grid = micropolygon mesh corresponding to contiguous surface region**
- **Reyes pipeline configuration defines**
  - **Target micropolygon area (typically 1/4 to 1 pixels)**
  - **Maximum number of micropolygons in a grid (typically ~256)**



Micropolygons



(note: here I'm showing triangle micropolygons, but for this lecture I usually refer to micropolygons as quads)

Today

- **Tessellation**
  - Lane-carpenter algorithm
- **Shading**
- **Hiding**
  - Stochastic rasterization
- **Transparency**
  - A-buffer algorithm

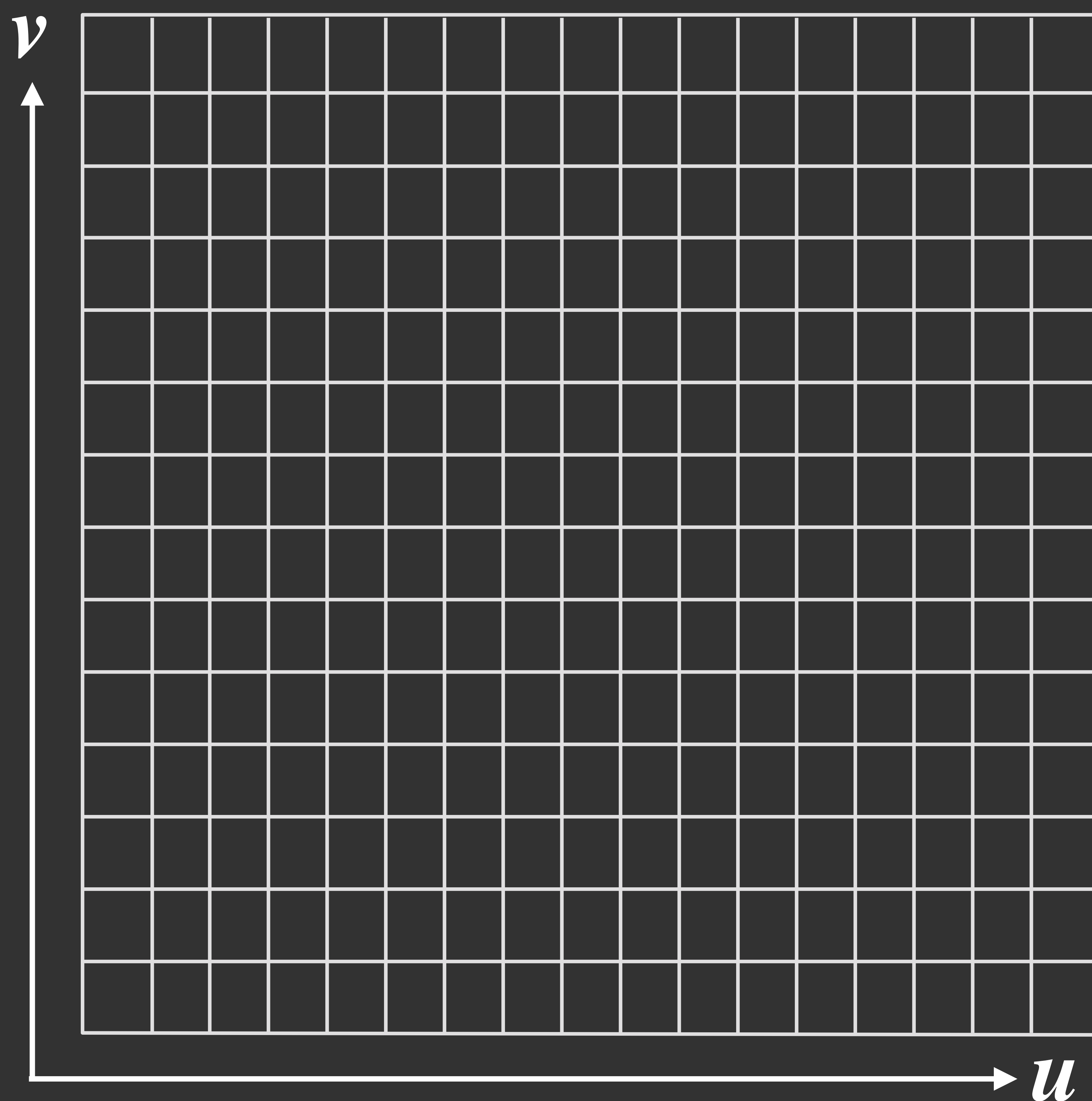
Tessellation

Tessellating primitives into micropolygon grids

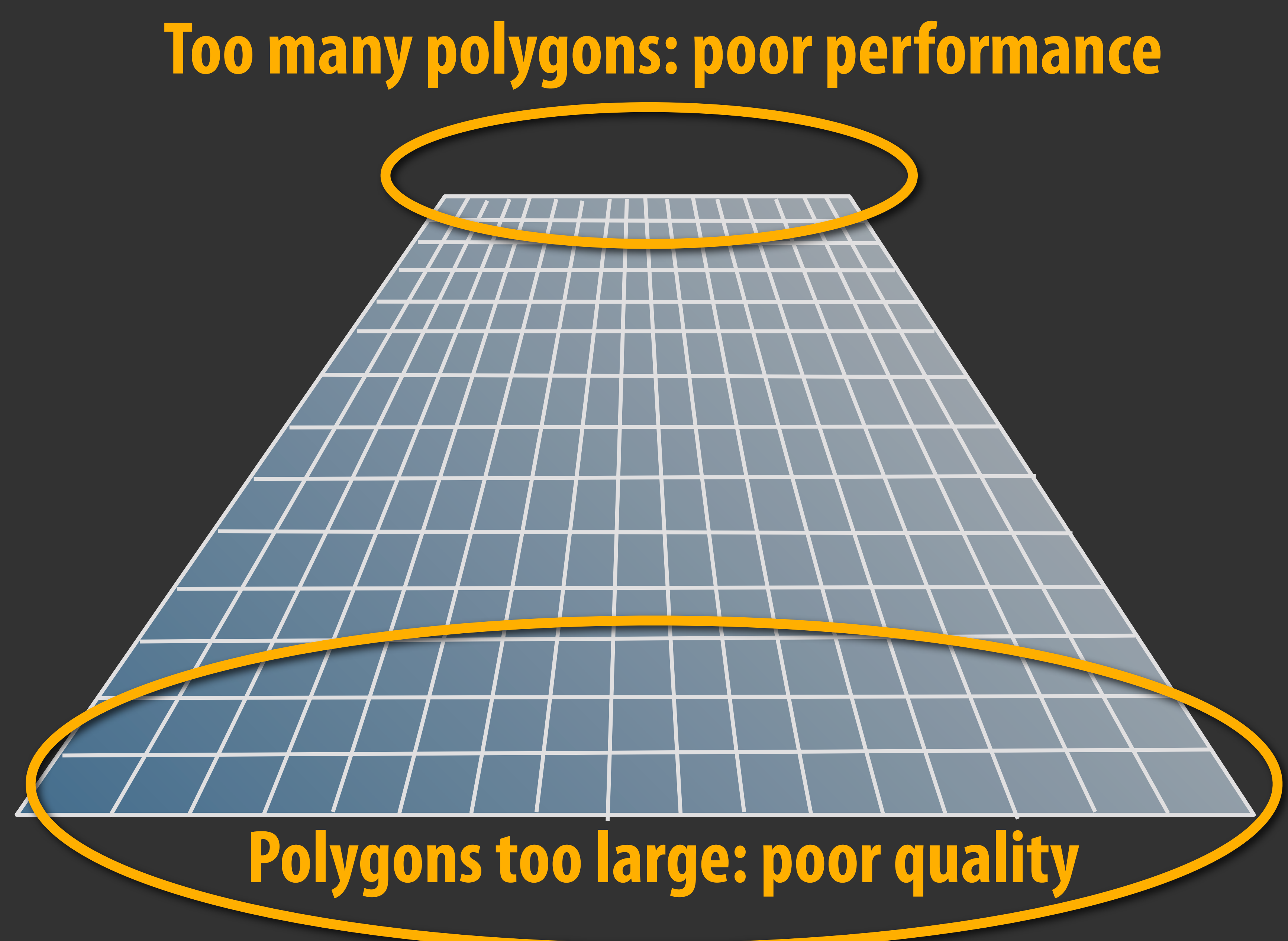
- **Goals**
  - Want micropolygons all about the same size
  - Want <u>projected</u> micropolygon areas to closely match target
  - Ideally, grids should be reasonably large (close to max grid size)

- **Reyes tessellation**
  - Lane-carpenter algorithm (often referred to as "split-dice")

Uniform patch tessellation is insufficient



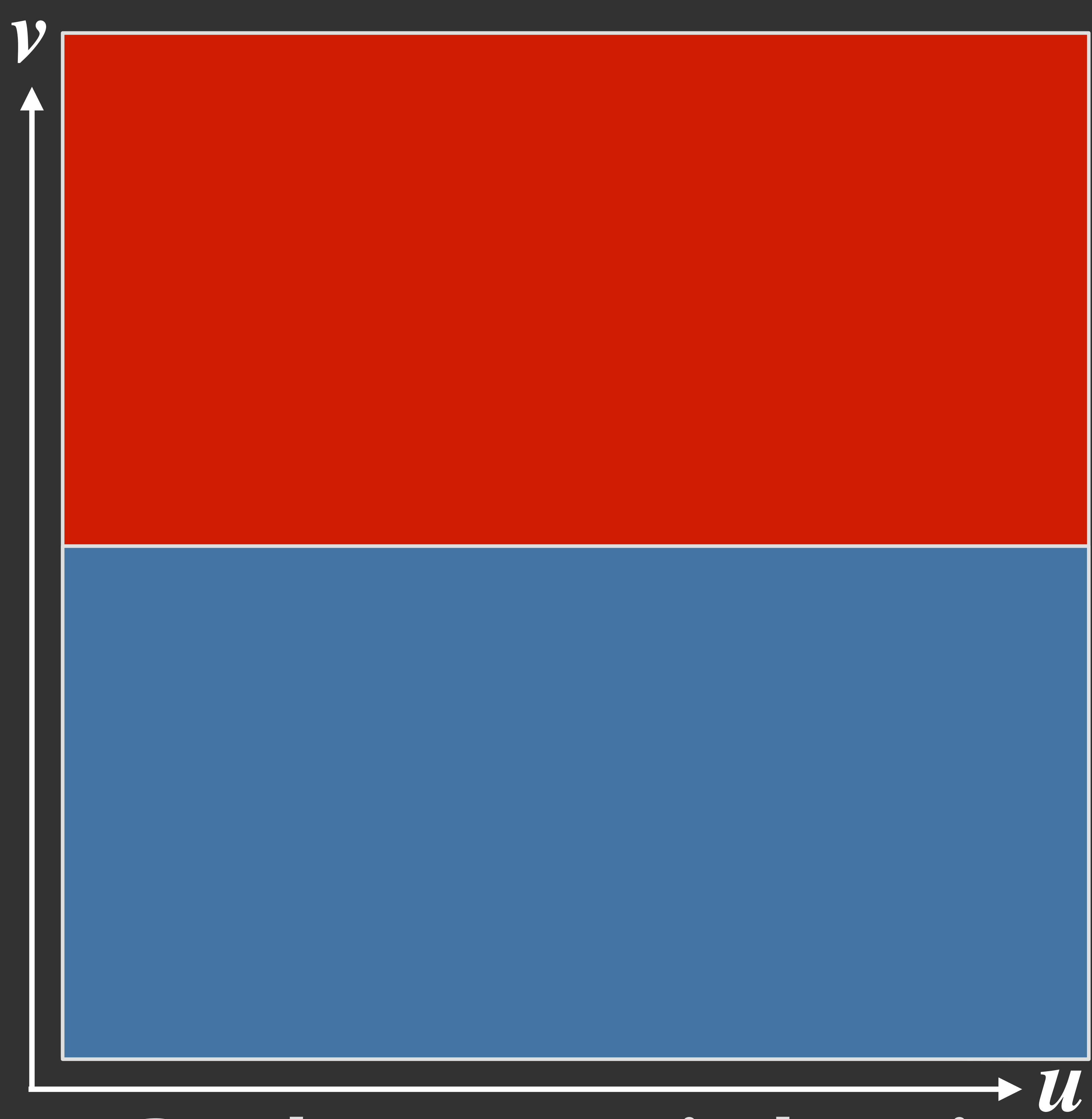
Uniform partitioning of patch (parametric domain)



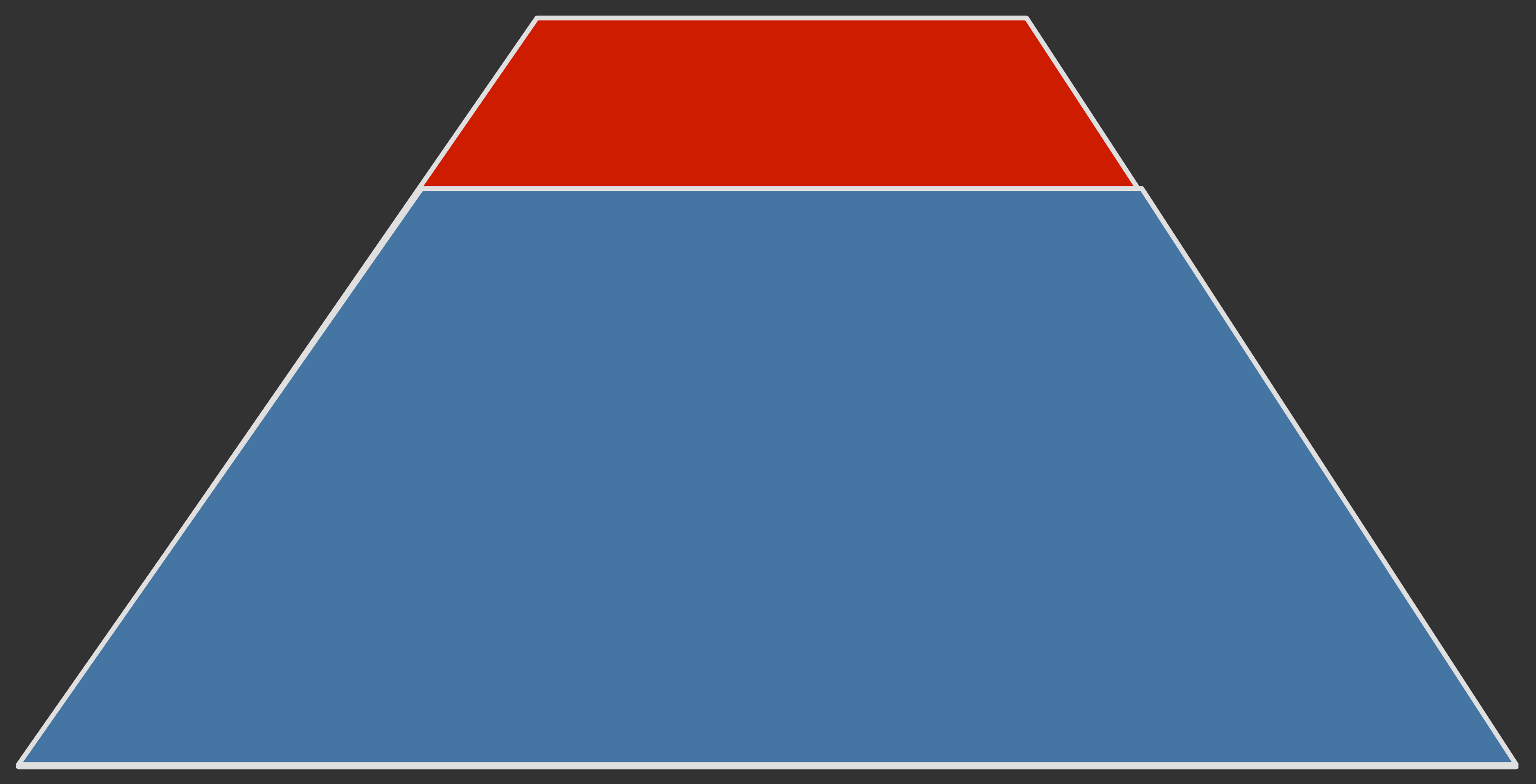
Patch viewed from camera

Split-dice adaptive tessellation

[Lane 80]



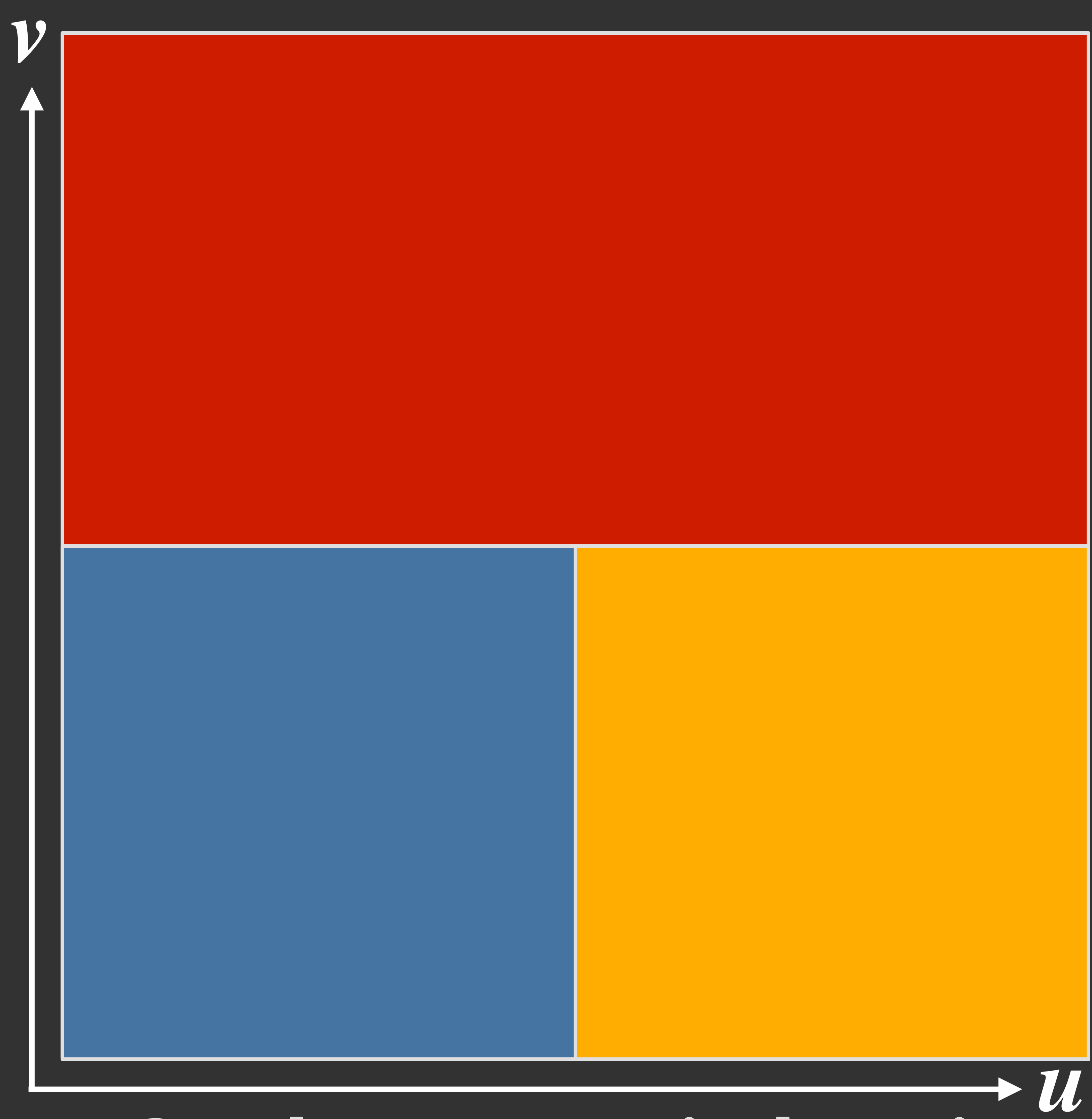
Patch parametric domain



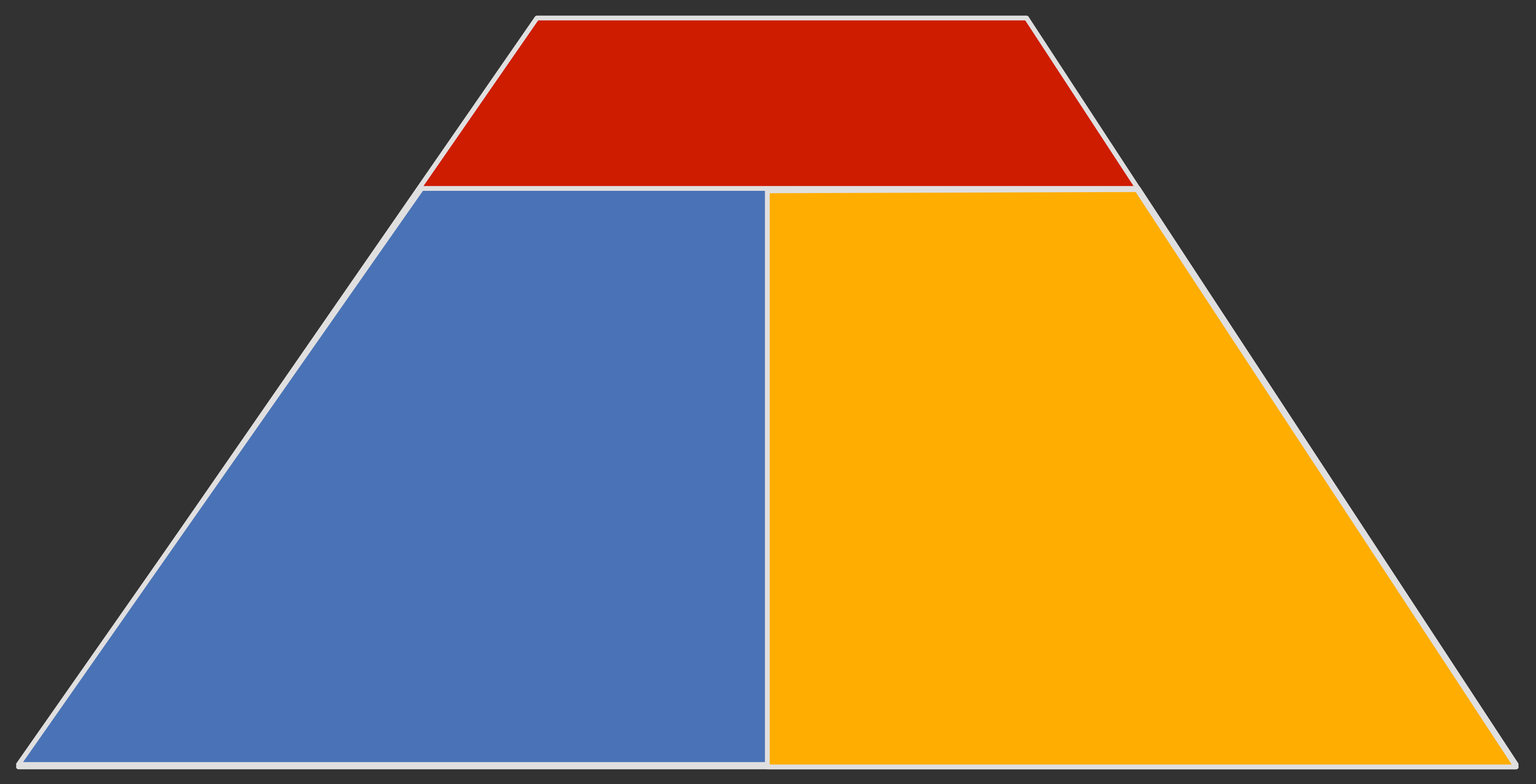
Patch viewed from camera

Split-dice adaptive tessellation

[Lane 80]



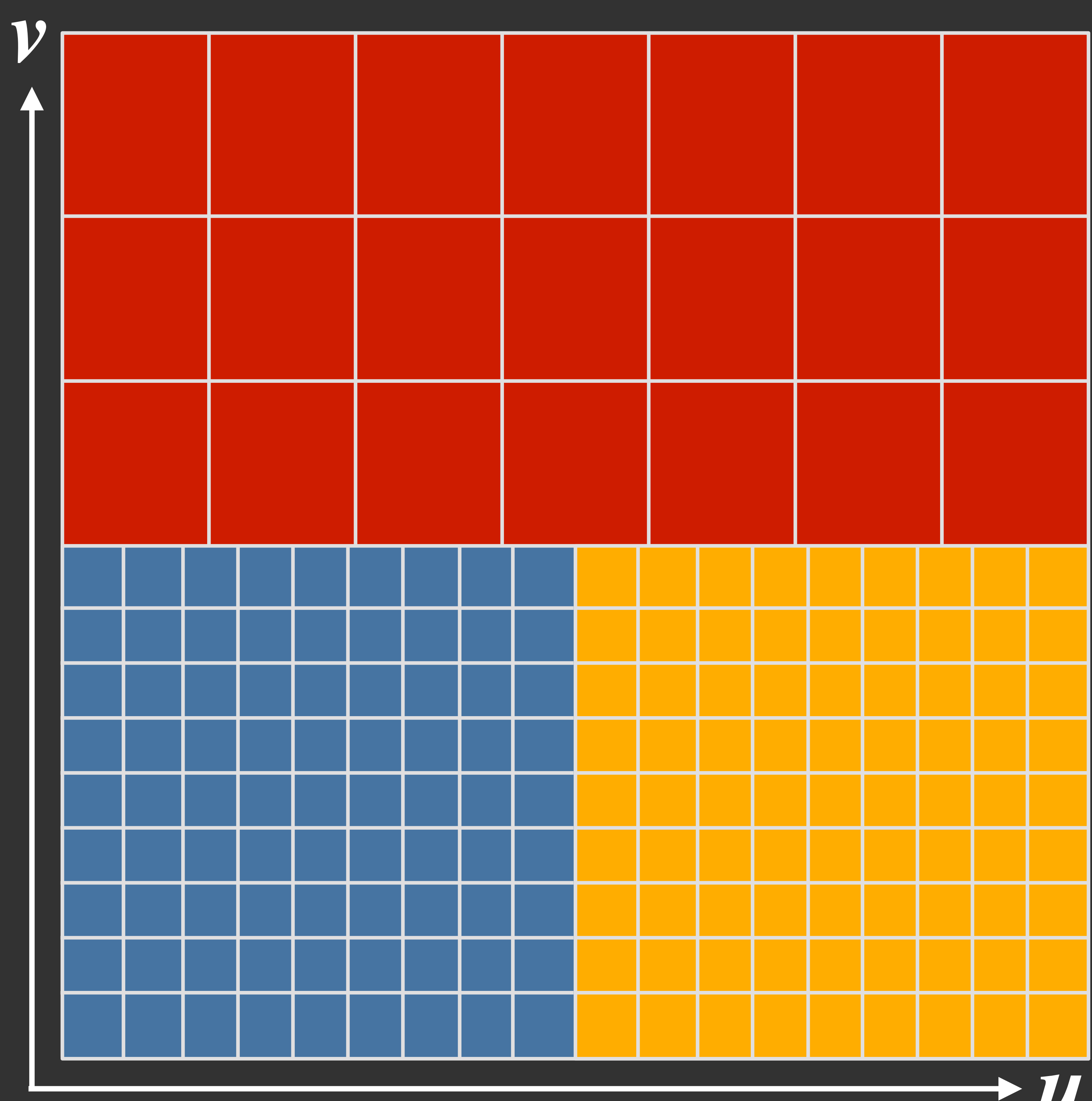
Patch parametric domain



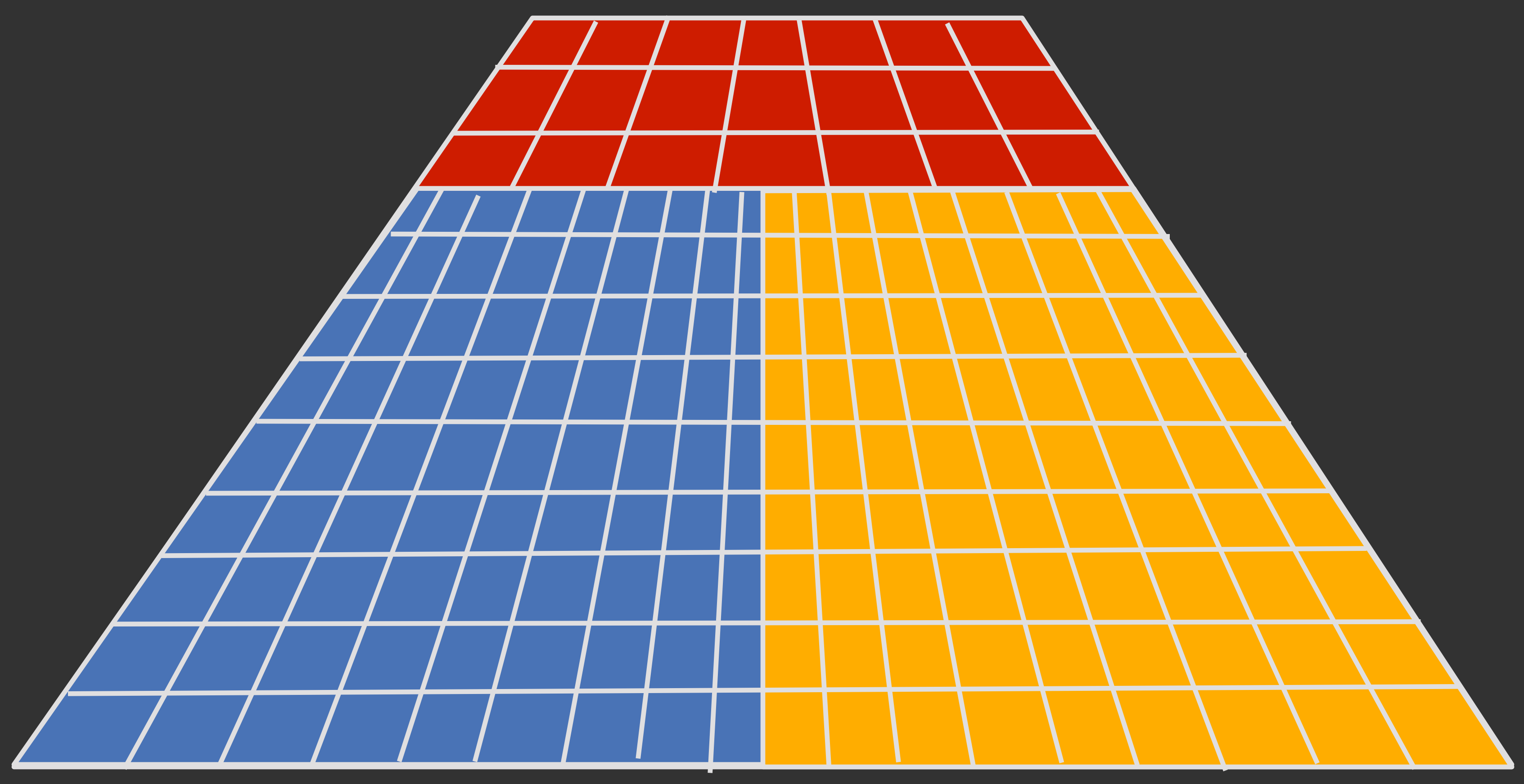
Patch viewed from camera

Split-dice adaptive tessellation

[Lane 80]



Patch parametric domain



Patch viewed from camera

Reyes primitive interface

```
class Primitive
{
   BBox3D          bbox();
   bool            canDice();
   List<Primitive> split();
   Grid            dice();
};
```

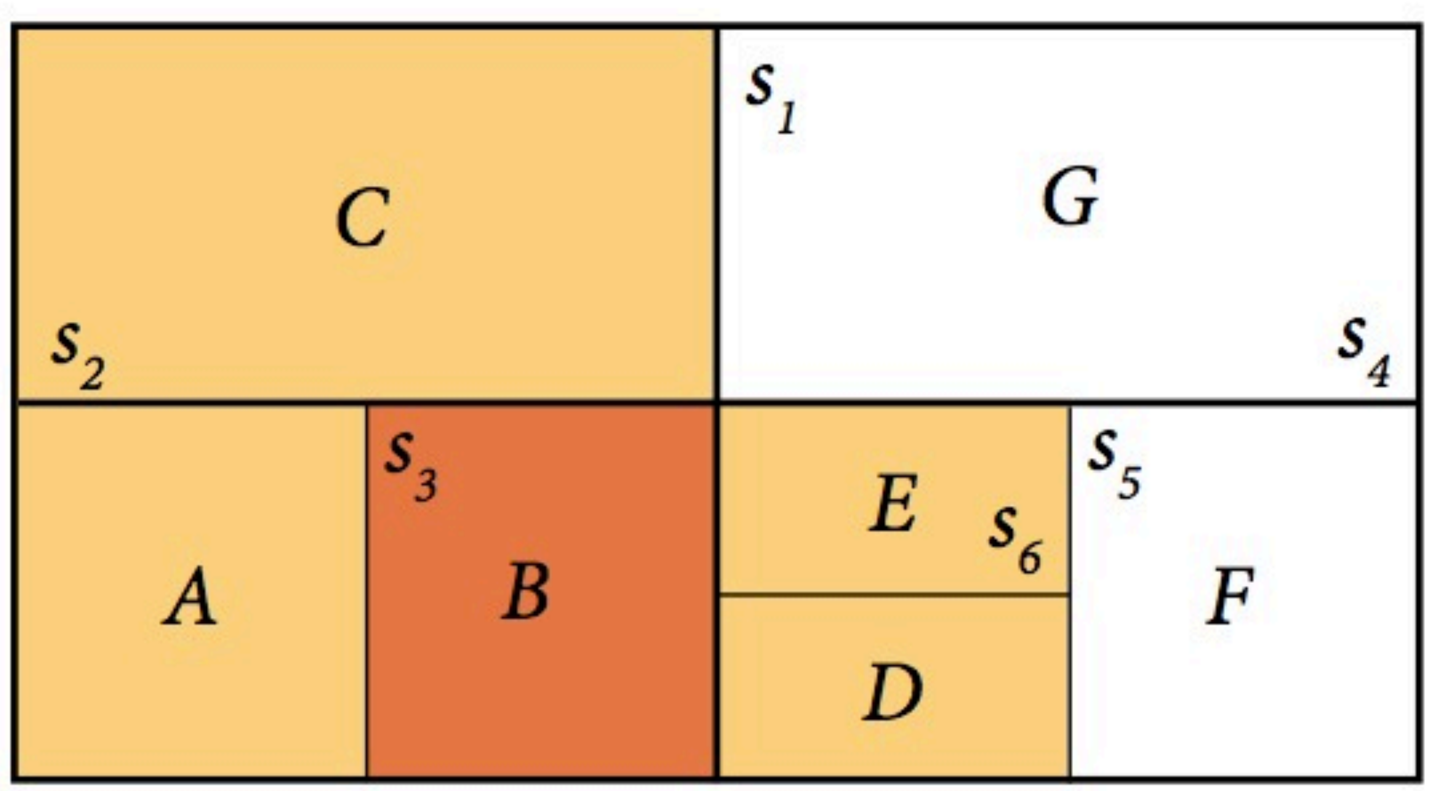
`Split` partitions primitive into 1 or more child primitives

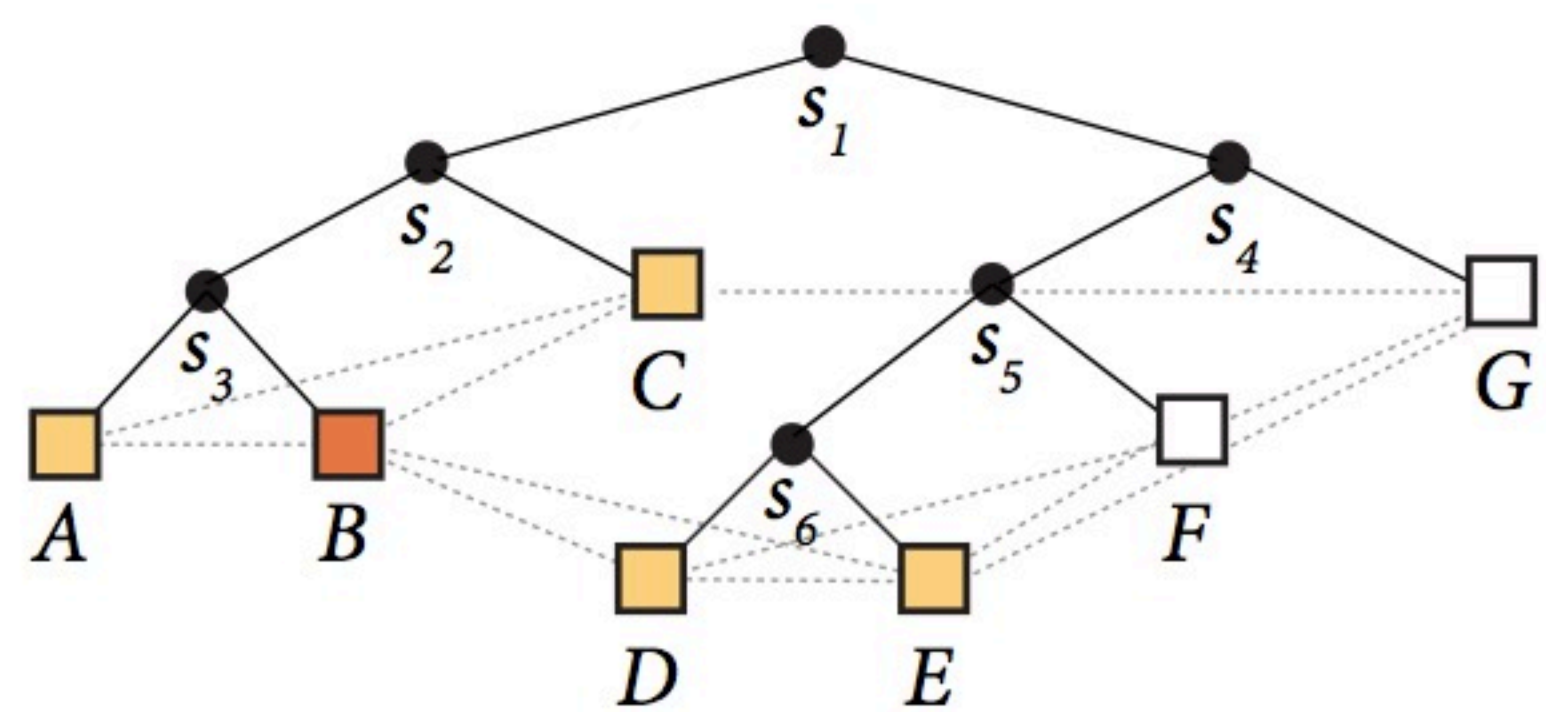
`Split` may generate child primitives of a different type

Note: bbox is expanded by renderer to account for primitive motion over the frame (motion blur), surface displacement, etc.

Interesting implications of split

- **Encapsulates adaptivity (keep dice simple, regular, and fast)**
- **Divide and conquer:**
  - Micropolygons generation order exhibits high spatial locality
  - Provides temporal stability
- **Splitting implicitly creates a hierarchy of grids**
  - Very useful for frustum/depth culling at largest possible granularity
  - Use bbox to cull primitives prior to dicing (or prior to unnecessary splitt





- **Splitting enables a clipless rasterization (see Reyes paper)**

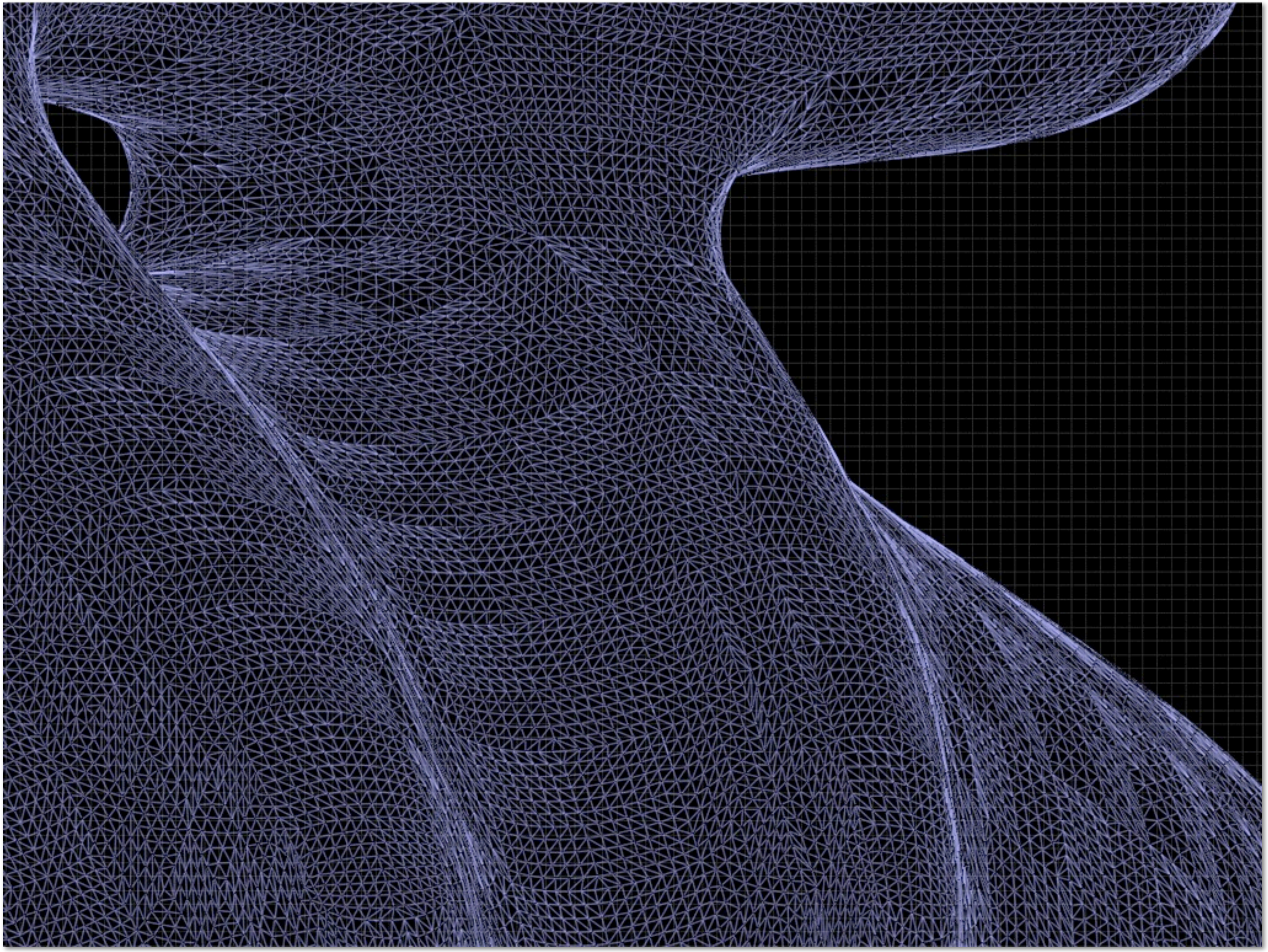
Shading

Reyes shades micropolygon grid vertices

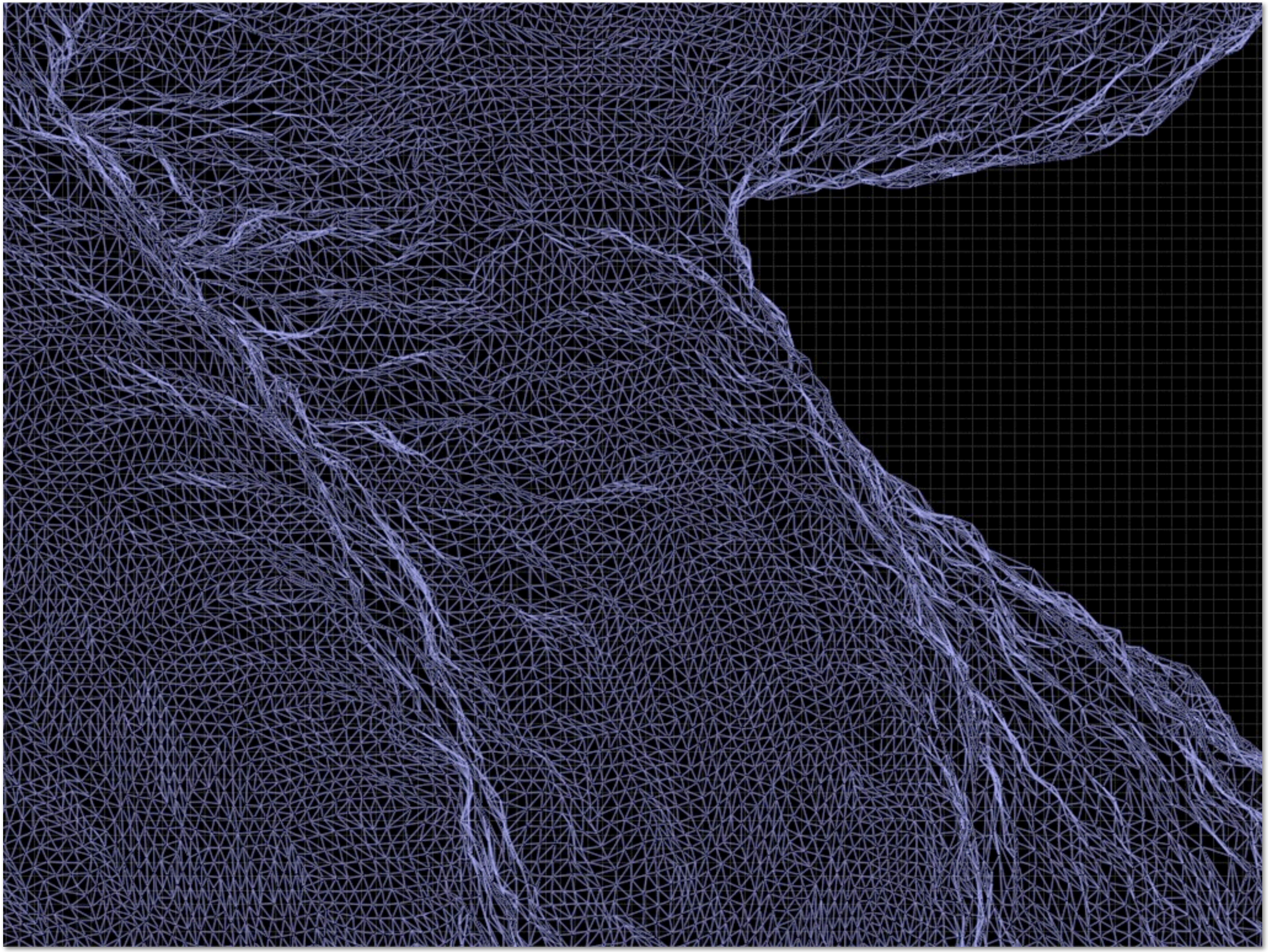
- **Reyes invokes the shading function once for each grid vertex**
  - Shading function defined using Renderman Shading Language (RSL) ***
  - Shading function computes surface appearance at vertex
  - Shading function may also reposition vertex (displacement)

*** See shading languages lecture

Micropolygon mesh: before displacement



Micropolygon mesh: after displacement



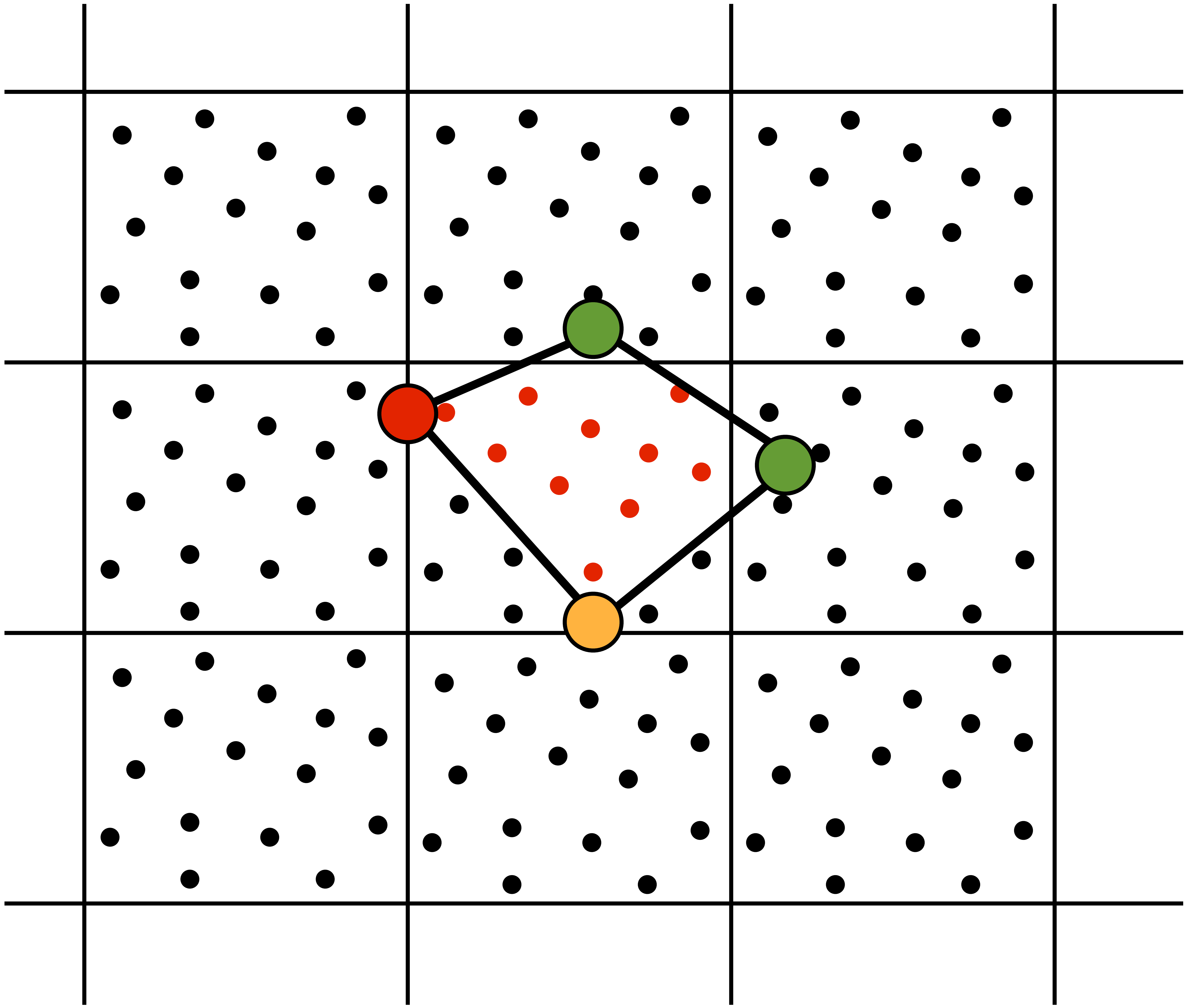
Why grids?

- **Execution coherence**
  - All vertices on grid shaded with same shader
  - Permits SIMD implementation
- **Locality**
  - Grid is contiguous region of surface: shading points together increases texture locality
- **Compact representation**
  - For regular (tensor product) grid, topology is implicit
  - Quad micropolygon grid: each interior vertex shared by four micropolygons
- **Connectivity leveraged to compute derivatives in shaders**
  - Can compute higher order derivatives
- **Preserve hierarchy**
  - Allows per-grid operations, in addition to per micropolygon or per-vertex
  - Useful for culling, etc.

Hiding

Hiding micropolygons (rasterization + occlusion)

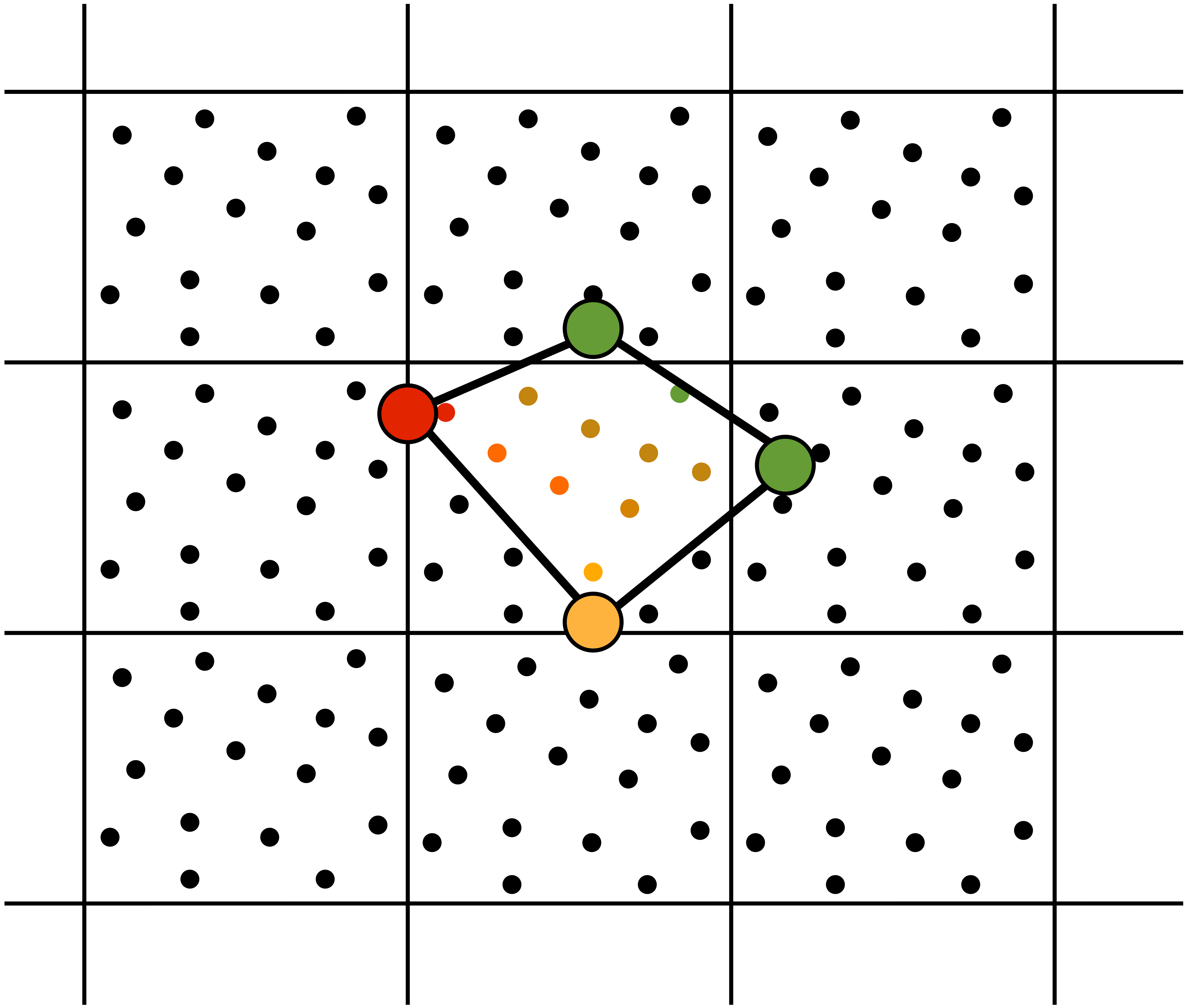
Option 1: micropolygon is flat shaded (apply color from one vertex to sample)



Note: many visibility samples per pixel to eliminate aliasing

Hiding micropolygons (rasterization + occlusion)

Option 2: interpolate per-vertex colors



Note: many visibility samples per pixel to eliminate aliasing

Aside: interesting sampling question

- Reyes samples surface appearance uniformly in parametric space (within in grid)
  - Uniform in parametric space ≃ uniform in object space, but not uniform in screen space due to projection
  - Textures filtered using object-space surface derivatives

- Surface is projected, and then appearance is <u>resampled</u> uniformly in screen space at visibility sample points

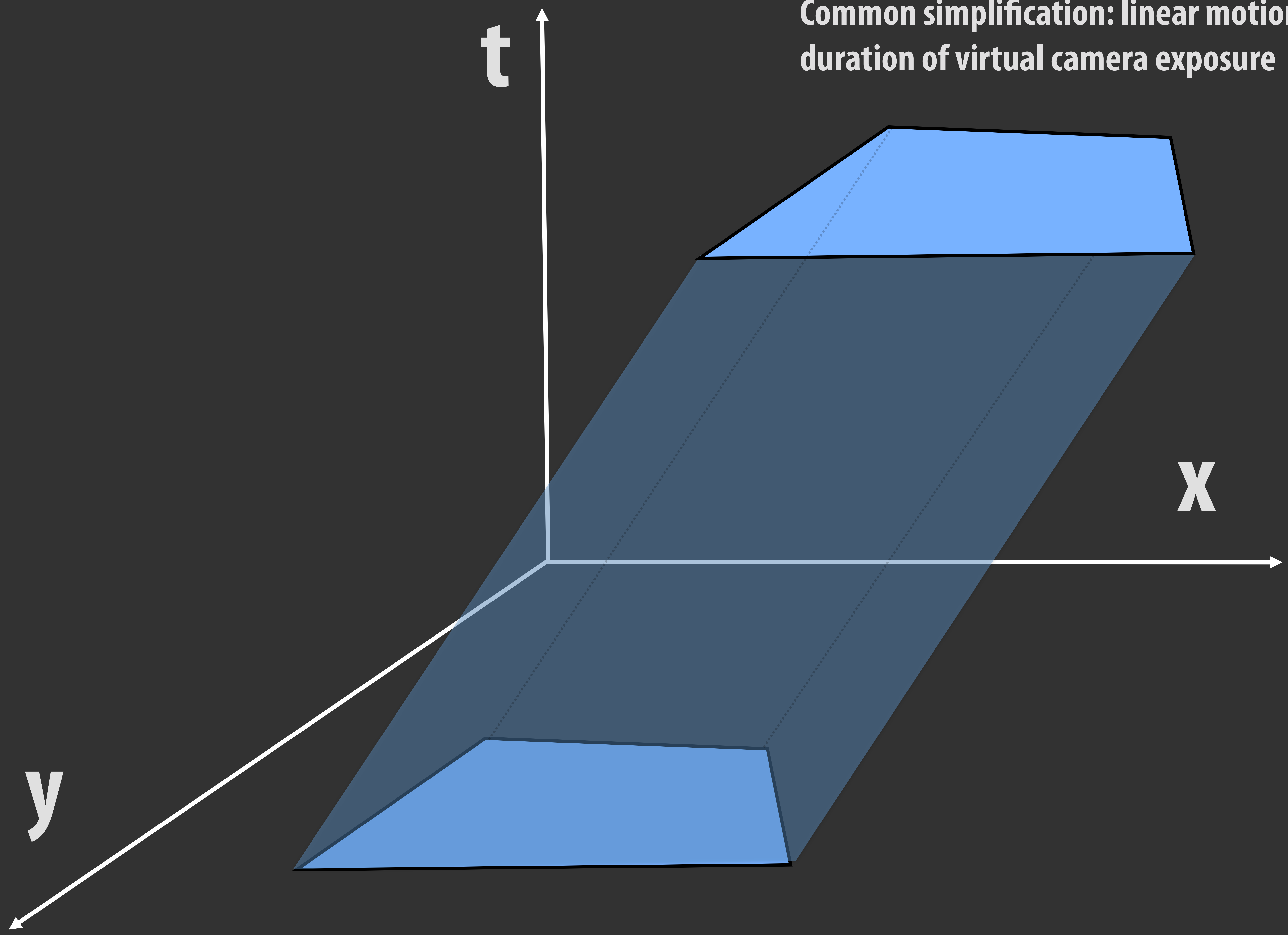
- OpenGL/Direct3D pipeline samples surface appearance uniformly in screen space
  - Textures filtered using screen-space surface derivatives

Question: is there a preferred solution?

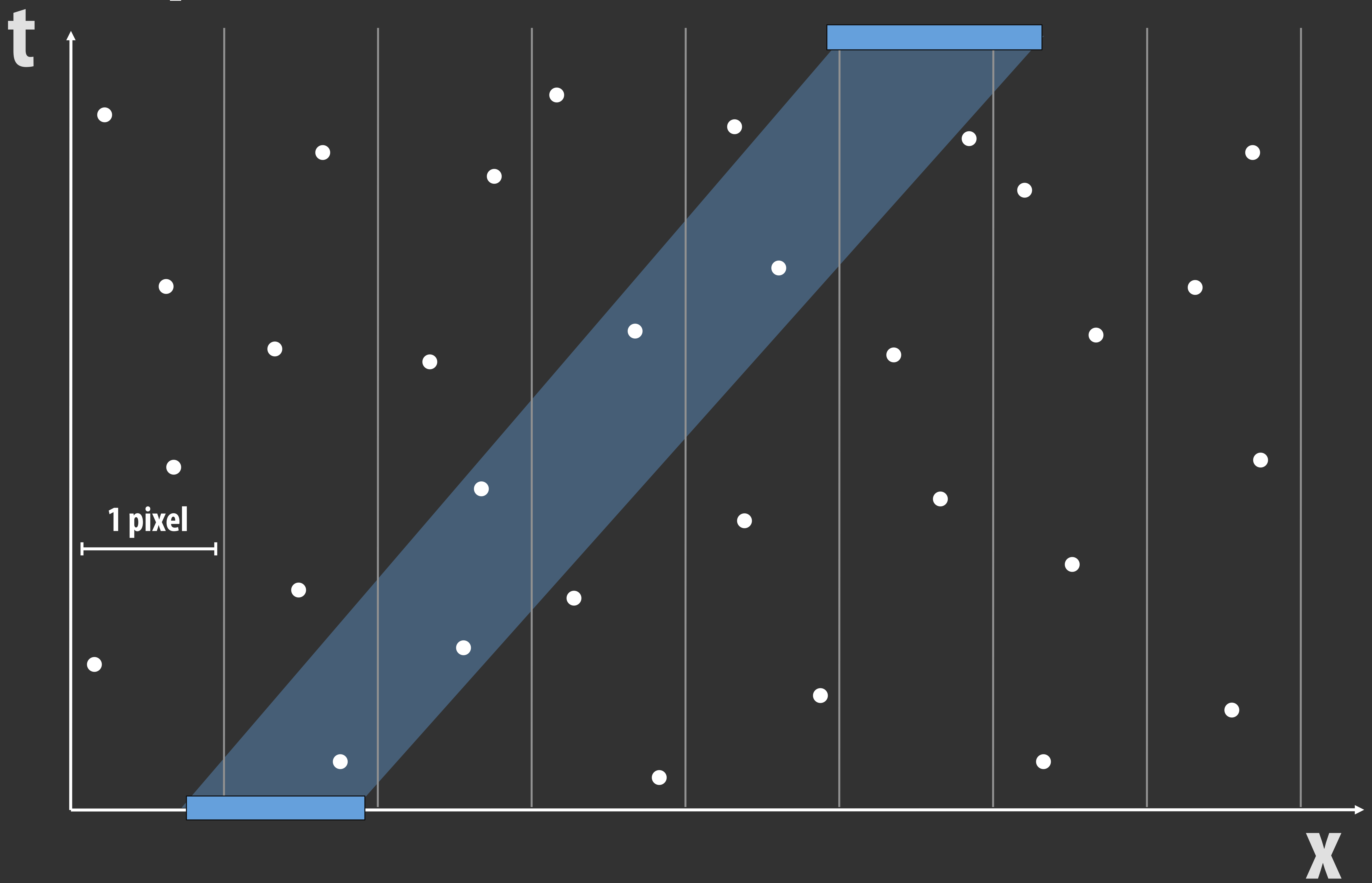
Consider:
High frequency surface appearance: due to bumpy geometry, due to high frequency texture
Surfaces at grazing angles to camera (near silhouettes)
What is lost in resampling step?

Moving micropolygon

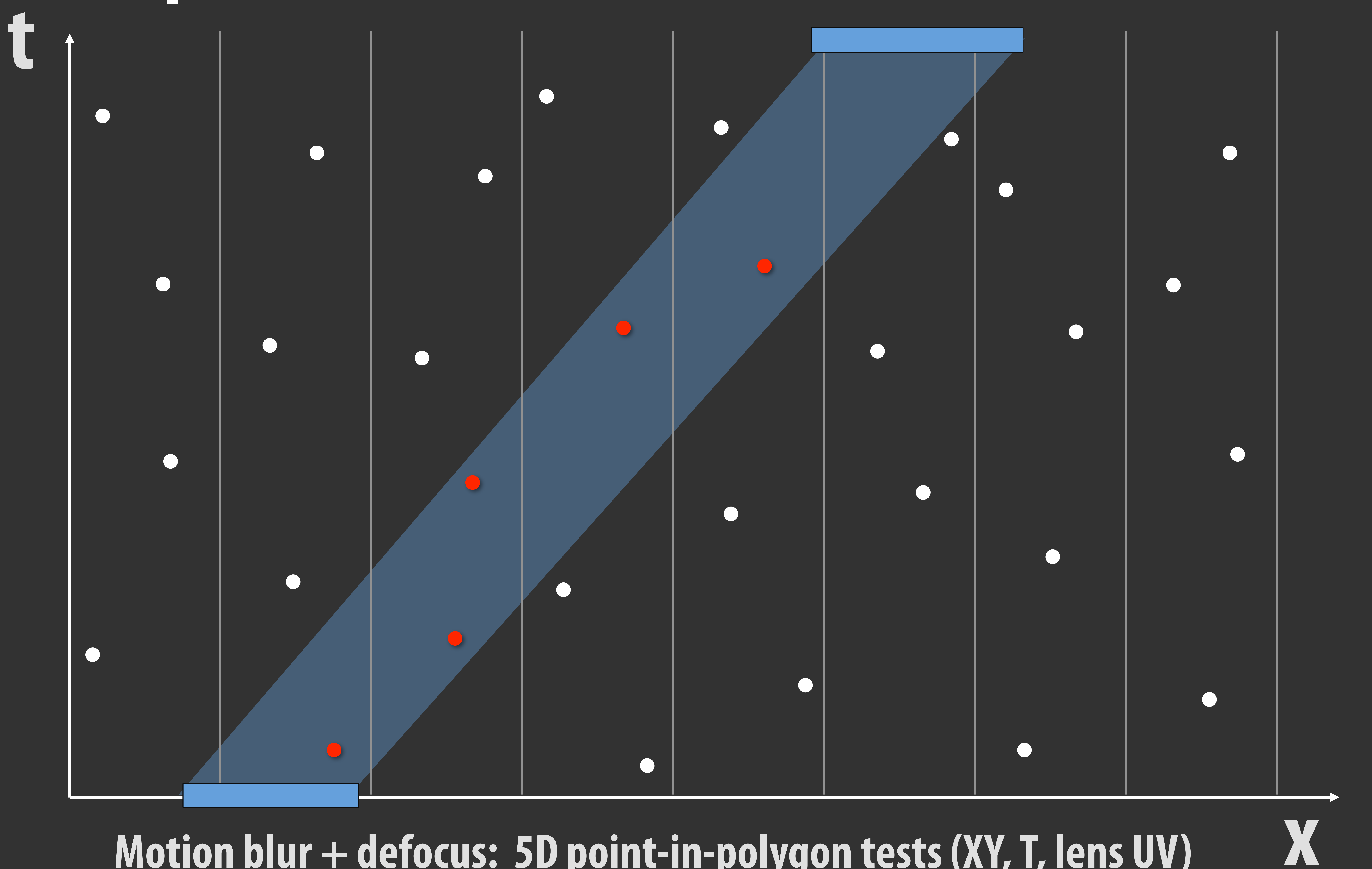
Common simplification: linear motion for duration of virtual camera exposure



X,T plane (visibility samples distributes in space and time)

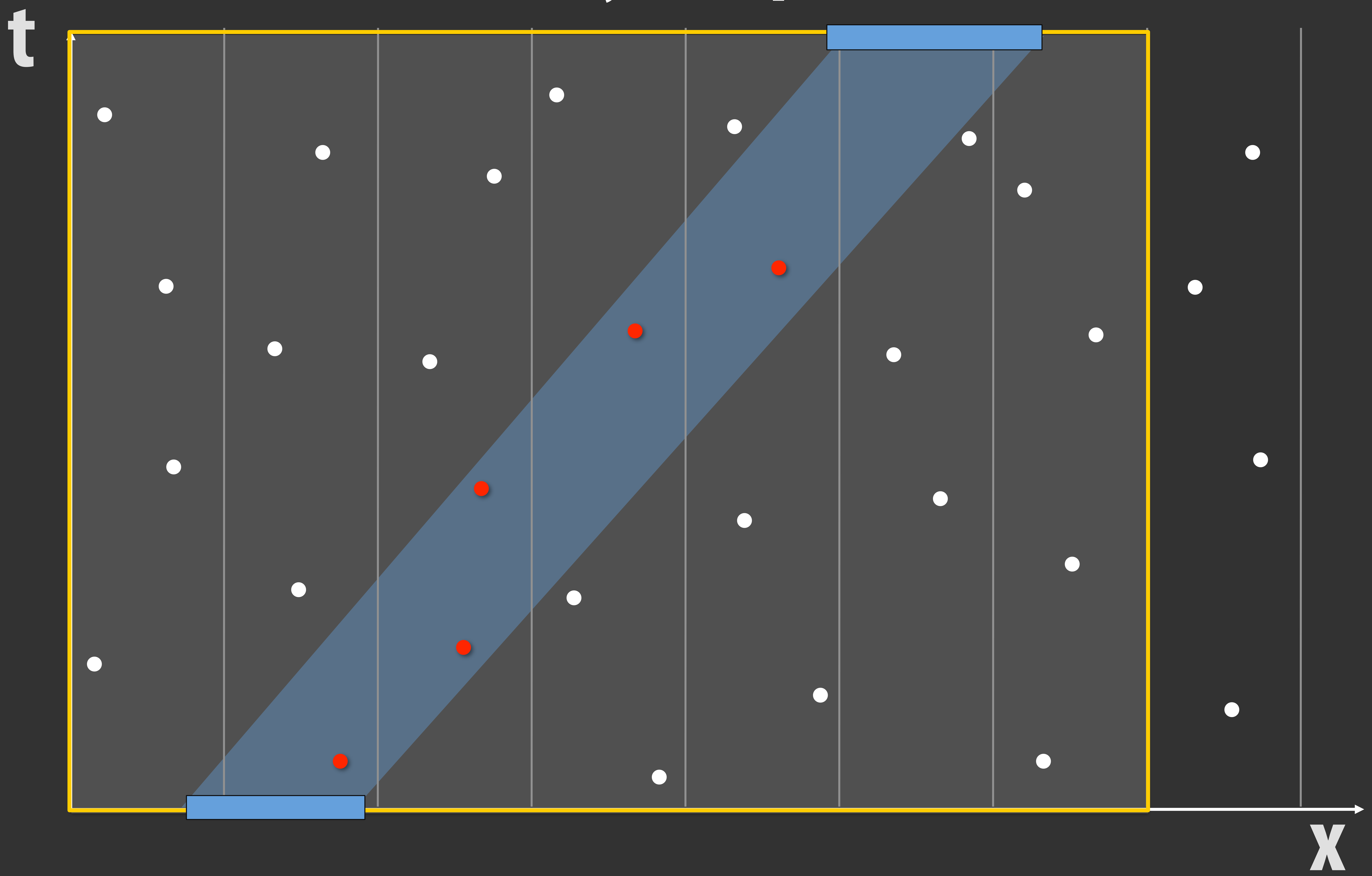


X,T plane

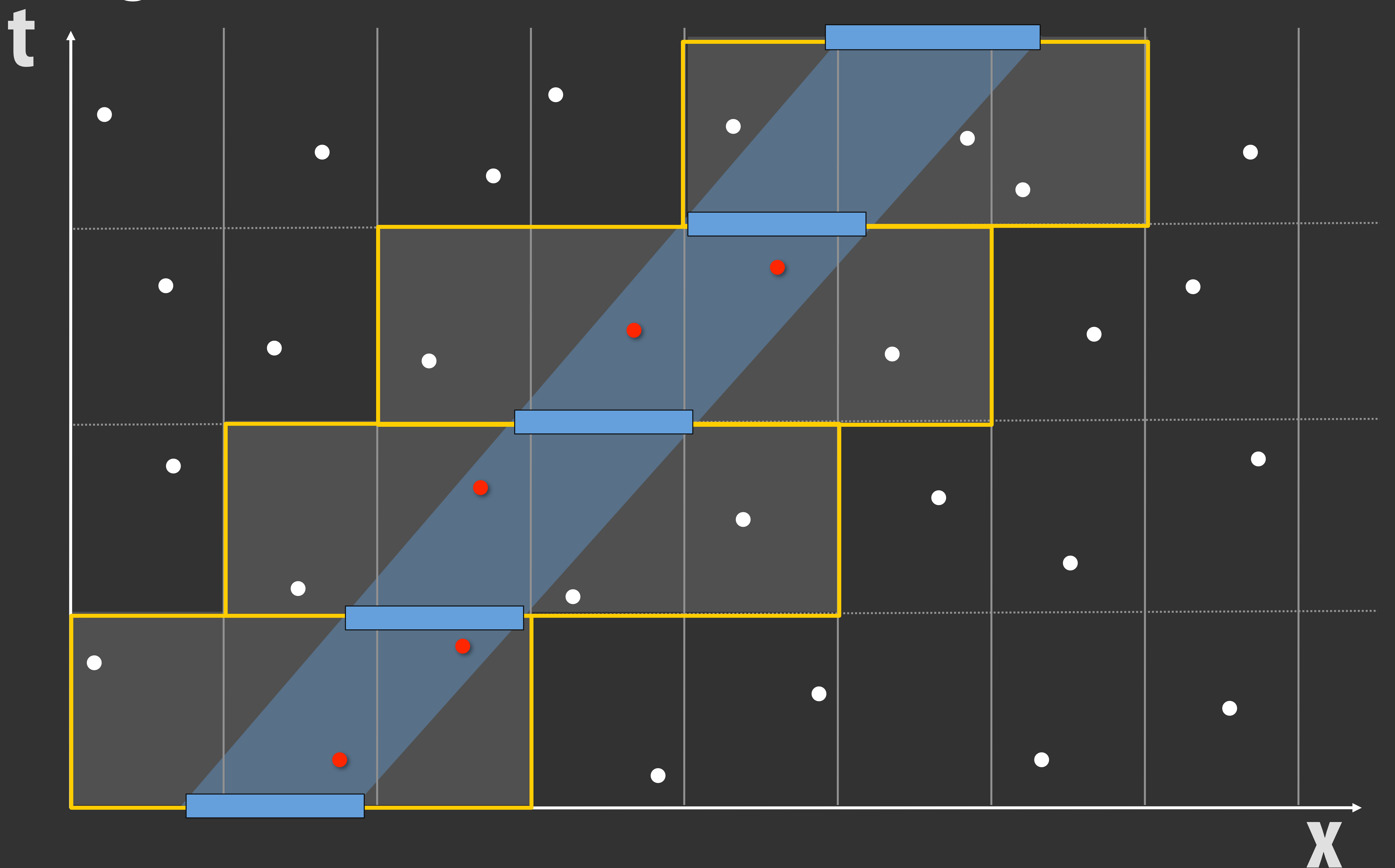


Motion blur + defocus: 5D point-in-polygon tests (XY, T, lens UV)

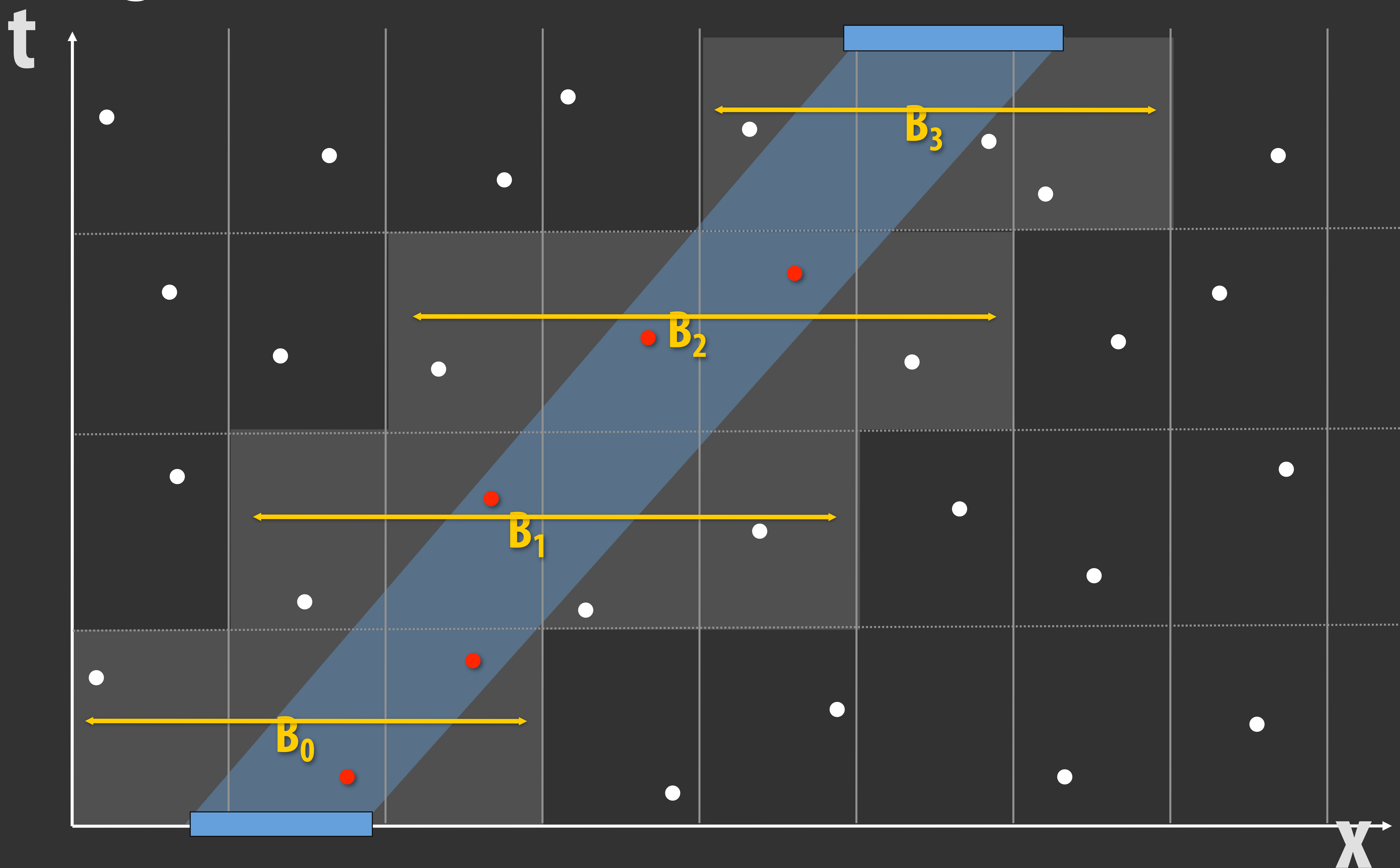
Candidate visibility samples



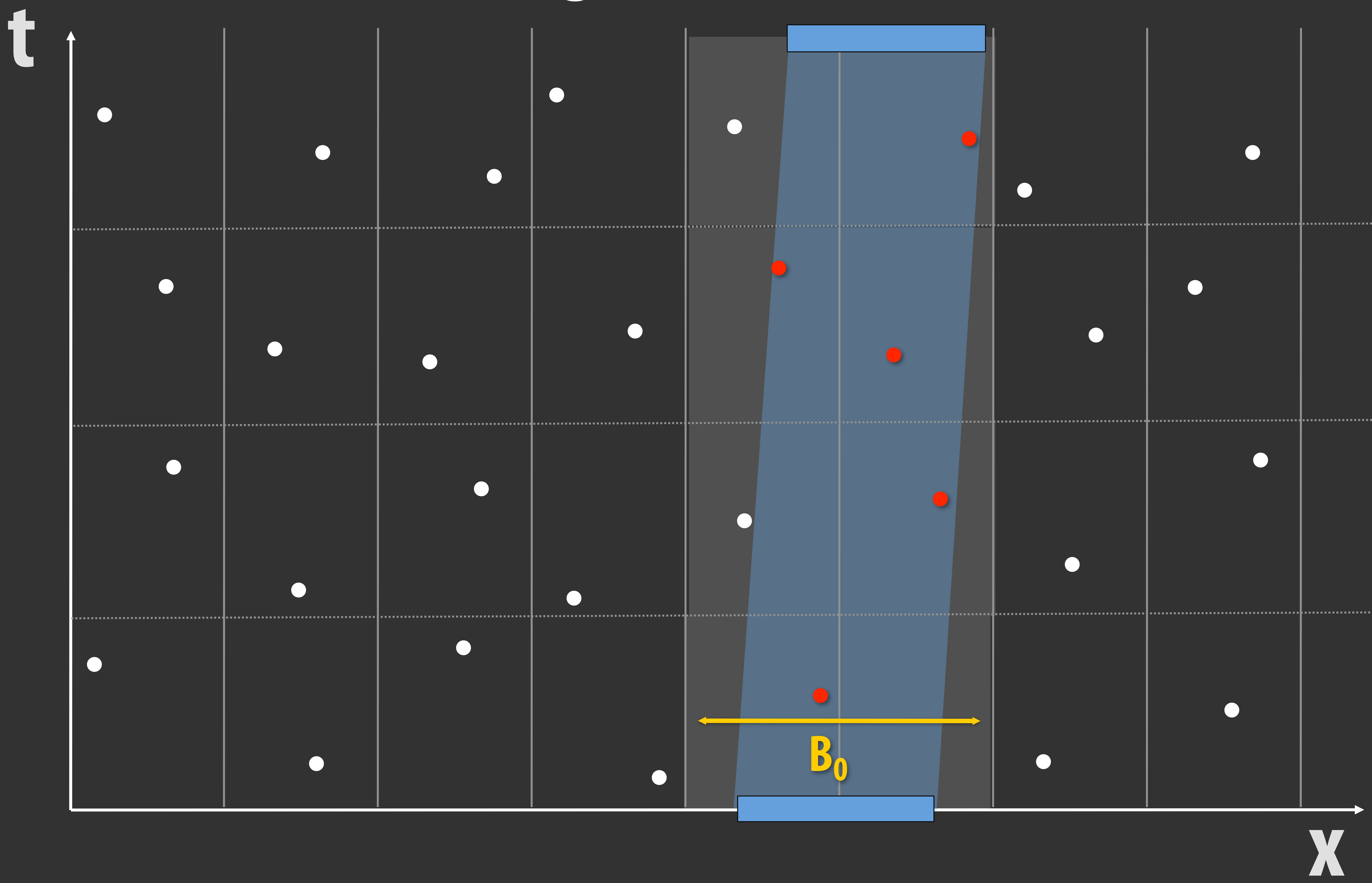
Tighter bounds (4 time intervals)



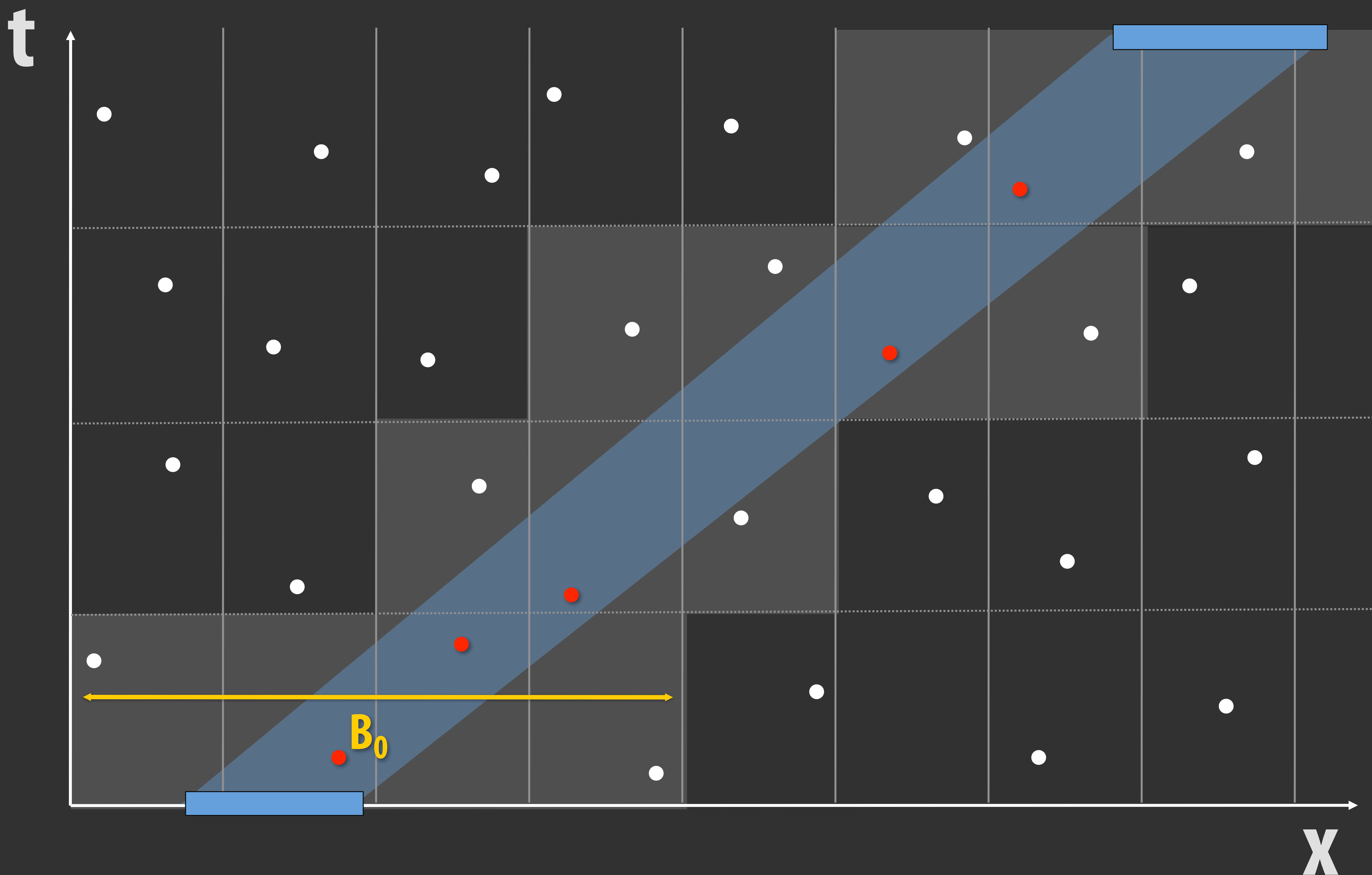
Tighter bounds (4 time intervals)



Slow motion = tight bounds

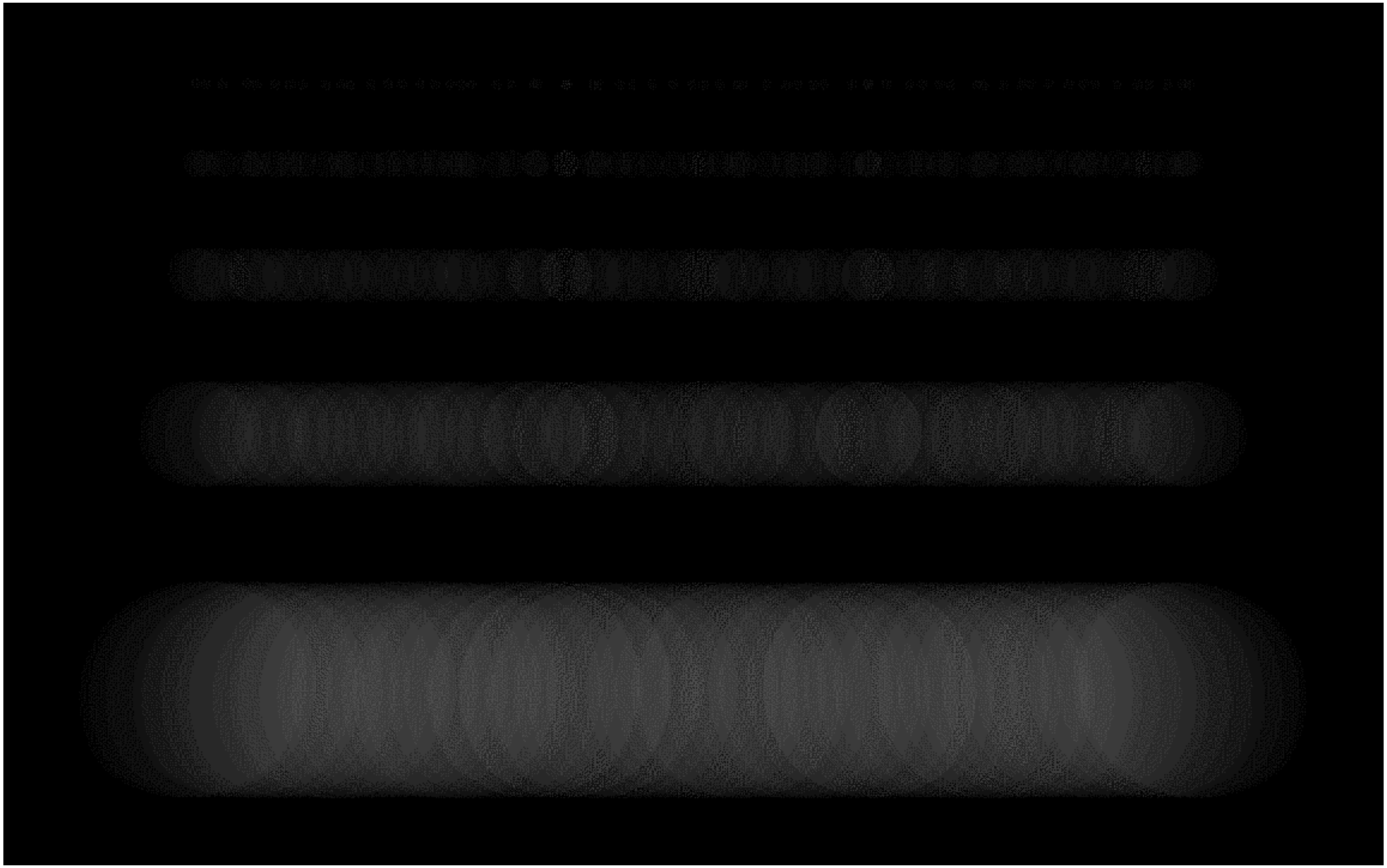


Fast motion = loose bounds



Stochastic rasterization results

White ball moving rapidly across screen
(movies shown in class)

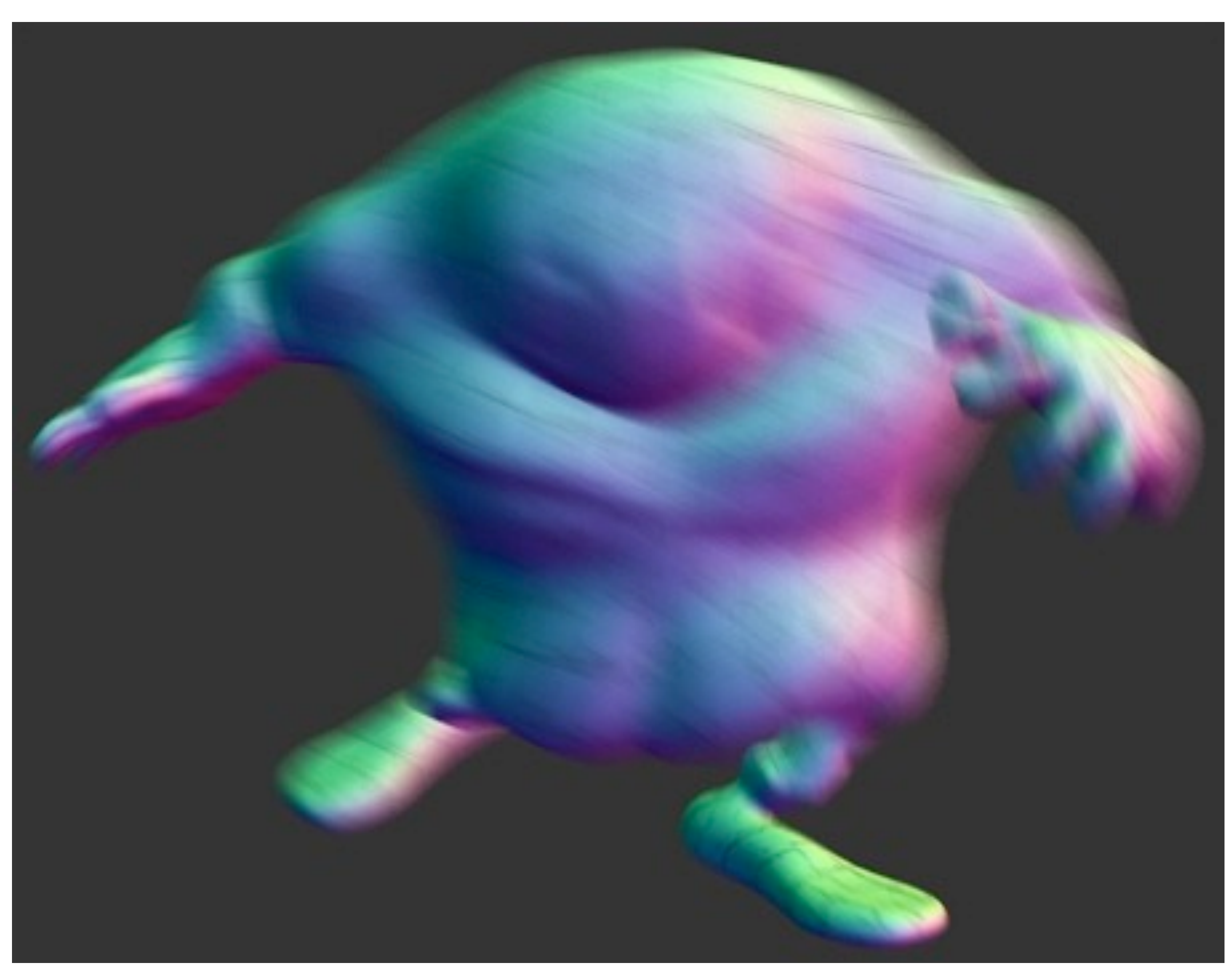


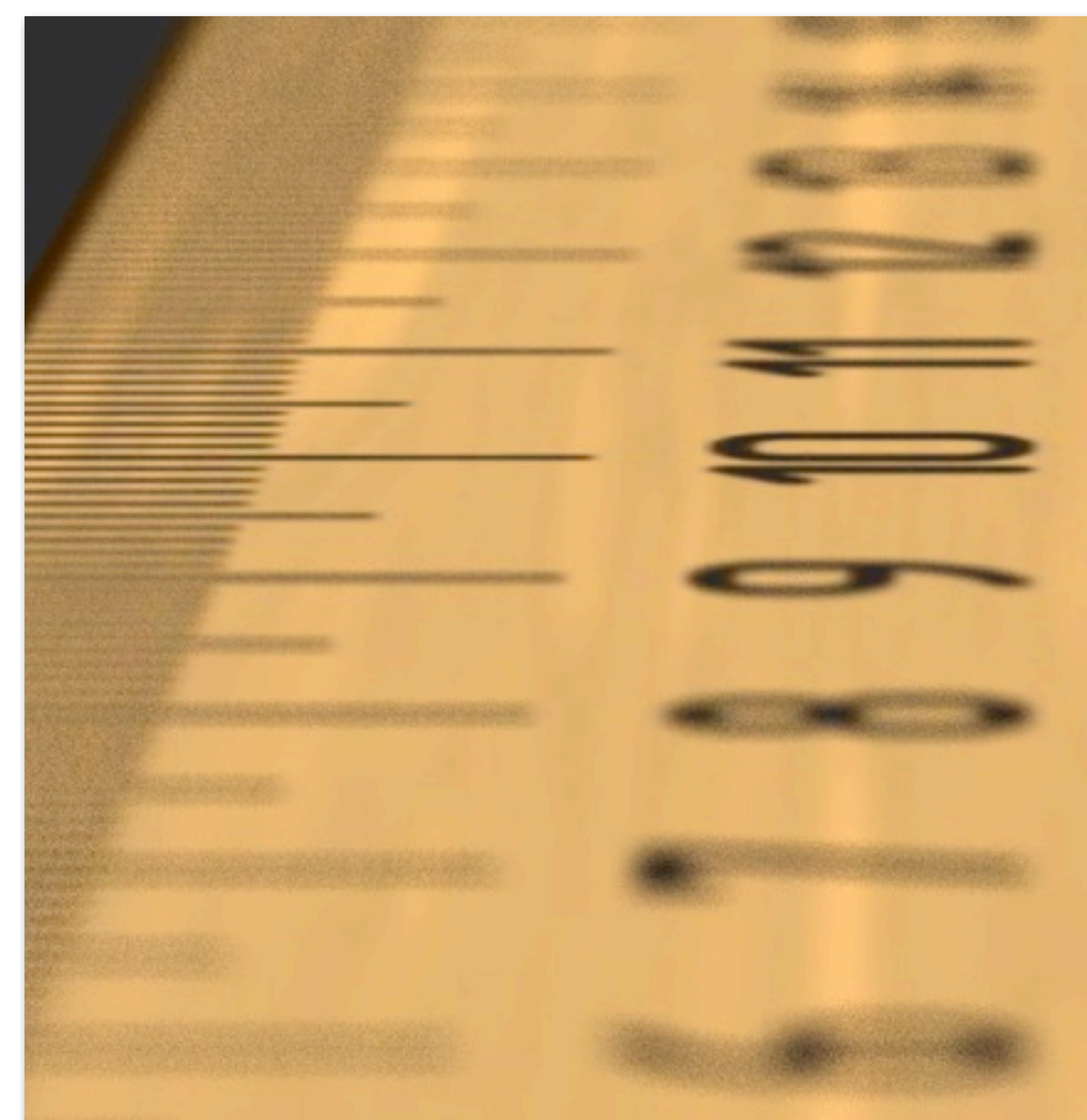
Stochastic rasterization results

White ball moving rapidly across screen
(movies shown in class: see web site)

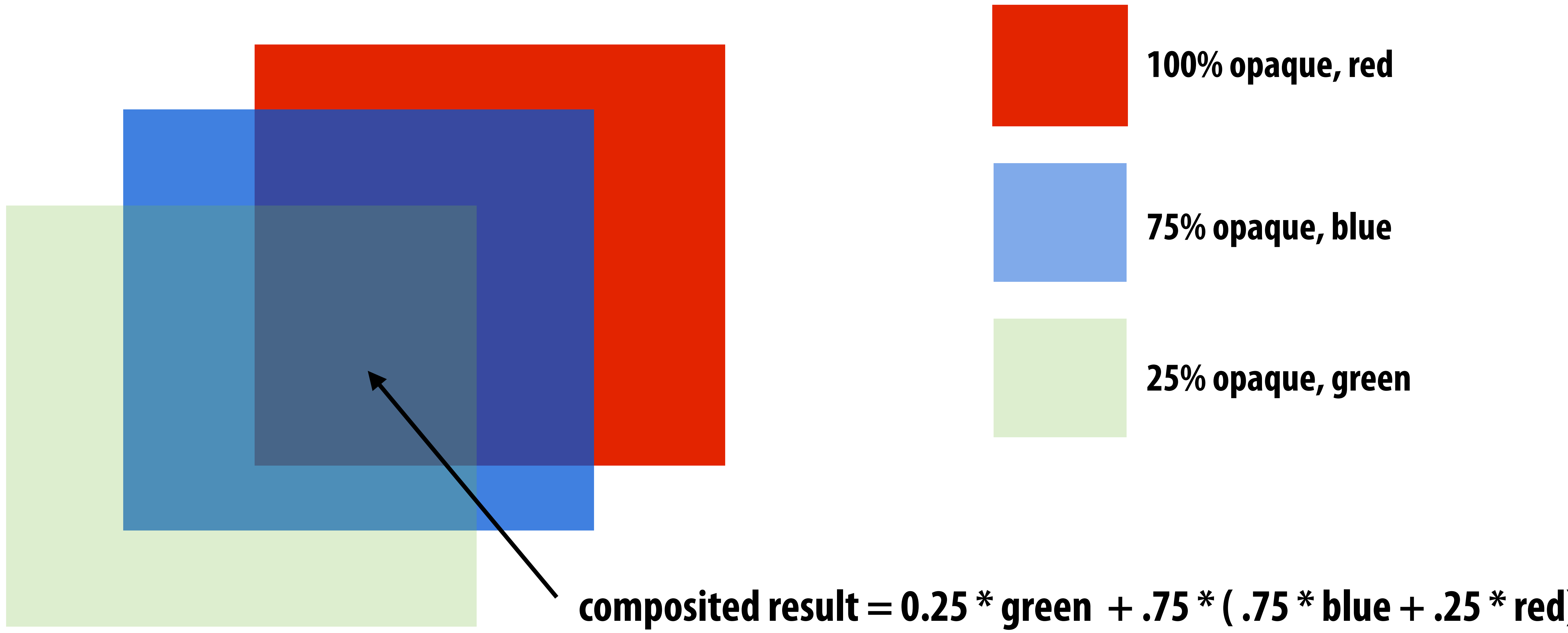
Stochastic sampling for motion blur (and defocus blur)

- Need high visibility sampling rates to remove noise in renderings with large motion blur, or camera defocus
- 64 - 128 visibility samples per pixel common in film rendering
  - Large frame-buffer!





Transparent surfaces



OpenGL/Direct3D solution relies on pipeline ordering semantics:
Application sorts surfaces, renders surfaces back-to-front ***
Set frame-buffer blend mode:

```
frag.alpha * frag.color + (1-frag.alpha) * fb_color
```

*** front-to-back rendering solution exists as well

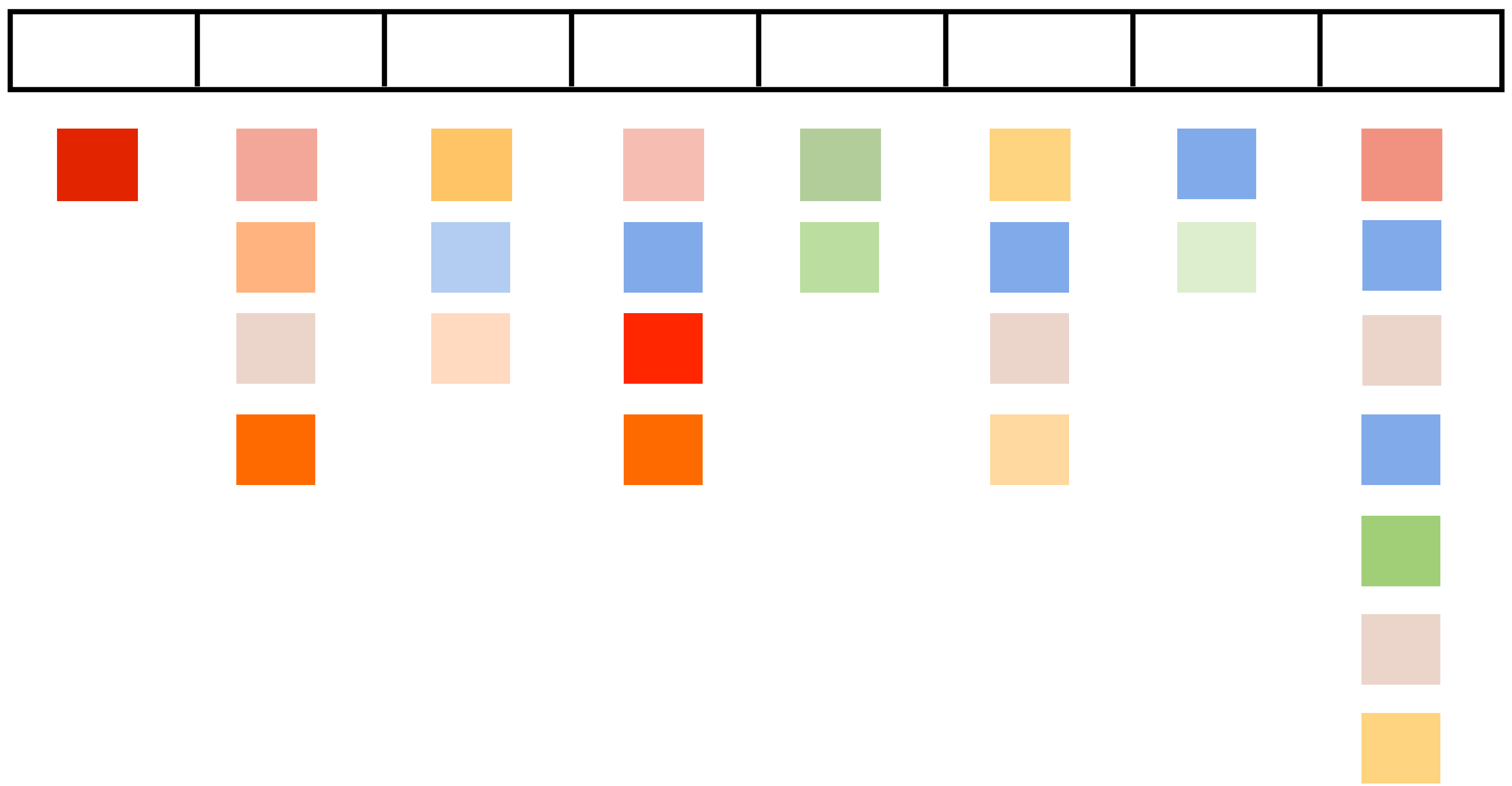
Transparency when using Z-buffer for occlusion

- Application sorting is a pain
- Depth sort order not well defined with triangles (interpenetration), let alone complex Reyes primitives
- Further complicated by motion blur

A-buffer

[Carpenter 84]

- Store list of "visible points" at each visibility sample
  - visible point = {rgb, alpha, z}



- When frame rendering is complete:

```
For each sample:
   Sort visible points in list by Z
   Blend front-to-back (or back-to-front)
```

- Provides primitive order-independent solution for rendering transparency
- Cost: variable storage per visibility sample
- Many optimizations to prune list as rendering proceeds
  - e.g., don't need to add visible points behind an opaque point in the list

Reyes A-buffer

- Many visibility samples per pixel (recall: 64-128)
- Many visible points per sample (under conditions of significant transparency)

1920x1080 rendering (1080p)

64 visibility samples per pixel

4 visible points per sample (rgb,a,z)

~10 GB A-buffer !!!

Reyes implementations use bucketing

- Recall "sort middle tiled chunked"
- Motivation here is to keep the A-buffer for a bucket in memory

  (previously we discussed how some implementations of OpenGL use a similar sorting scheme to: gain parallelism, keep a tile of frame-buffer on chip)

```
for each primitive, place in screen bucket
for each bucket
    allocate G-buffer for bucket
    for each primitive
       split-dice to create grids // each split, cull primitives falling outside of bucket
       shade + hide grids
    for each bucket g-buffer sample
       composite visible points
```

Reyes summary

- **Key algorithms**
  - High quality, split-dice tessellation
  - Shades per-vertex, prior to rasterization
  - Visibility via stochastic point sampling to simulate motion blur, camera defocus
  - Correct rendering of transparent surfaces via the A-buffer

- **Key system concepts**
  - Micropolygons: common intermediate representation for all primitive types
  - Micropolygon grids for locality and SIMD shading
  - Bucketed rendering to fit tiles of A-buffer in memory (high depth complexity due to transparency and high visibility sampling rates)

    (not discussed today: lots of smarts in a performant Reyes implementation to keep working set in memory)