Lecture 12:
Deferred Shading

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

Special thanks to Andrew Lauritzen (Intel) and Johan Andersson (DICE) for producing excellent tutorials which influenced the content in this lecture

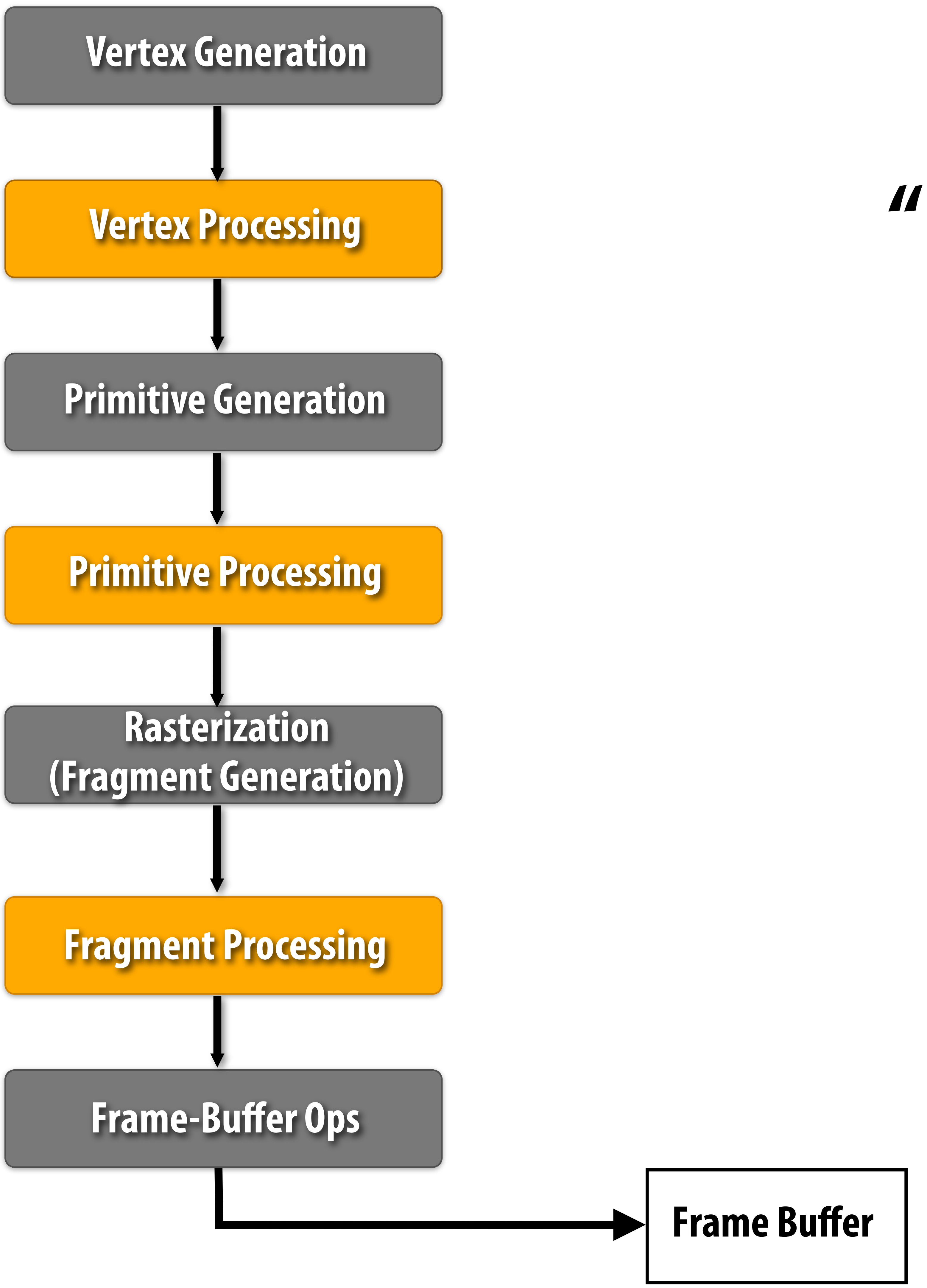
Today: deferred shading

- Idea: restructure the rendering pipeline to perform shading <u>after</u> all occlusions have been resolved

- Not a new idea: implemented in several old graphics systems, but not directly supported by modern graphics APIs and GPUs
  - [Deering et al. 88]
  - UNC PixelFlow [Molnar et al. 92]

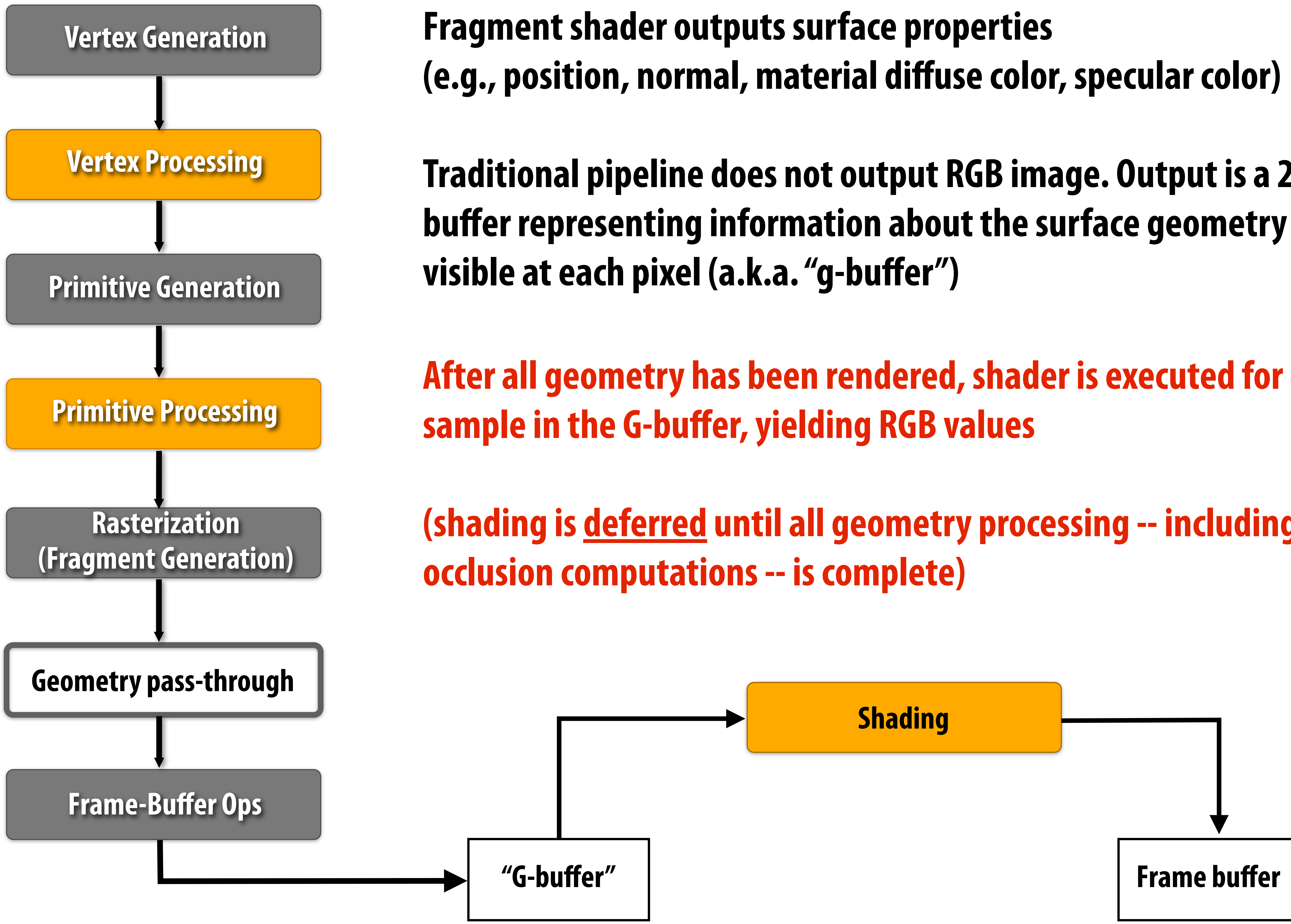
- Increasingly popular alternative algorithm for rendering

The graphics pipeline



"Forward rendering"

Deferred shading pipeline



Fragment shader outputs surface properties (e.g., position, normal, material diffuse color, specular color)

Traditional pipeline does not output RGB image. Output is a 2D buffer representing information about the surface geometry visible at each pixel (a.k.a. "g-buffer")

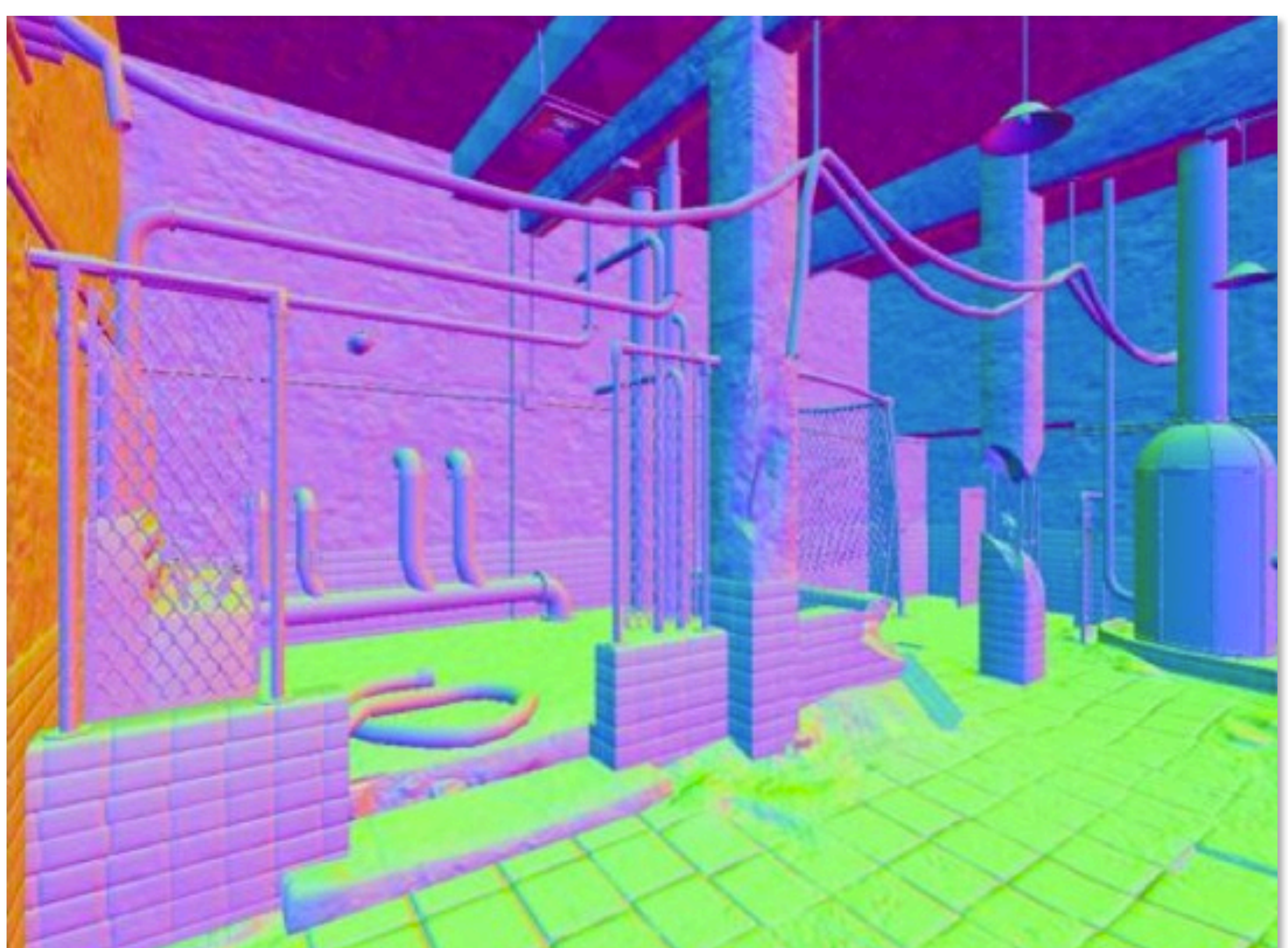
After all geometry has been rendered, shader is executed for each sample in the G-buffer, yielding RGB values

(shading is <u>deferred</u> until all geometry processing -- including all occlusion computations -- is complete)

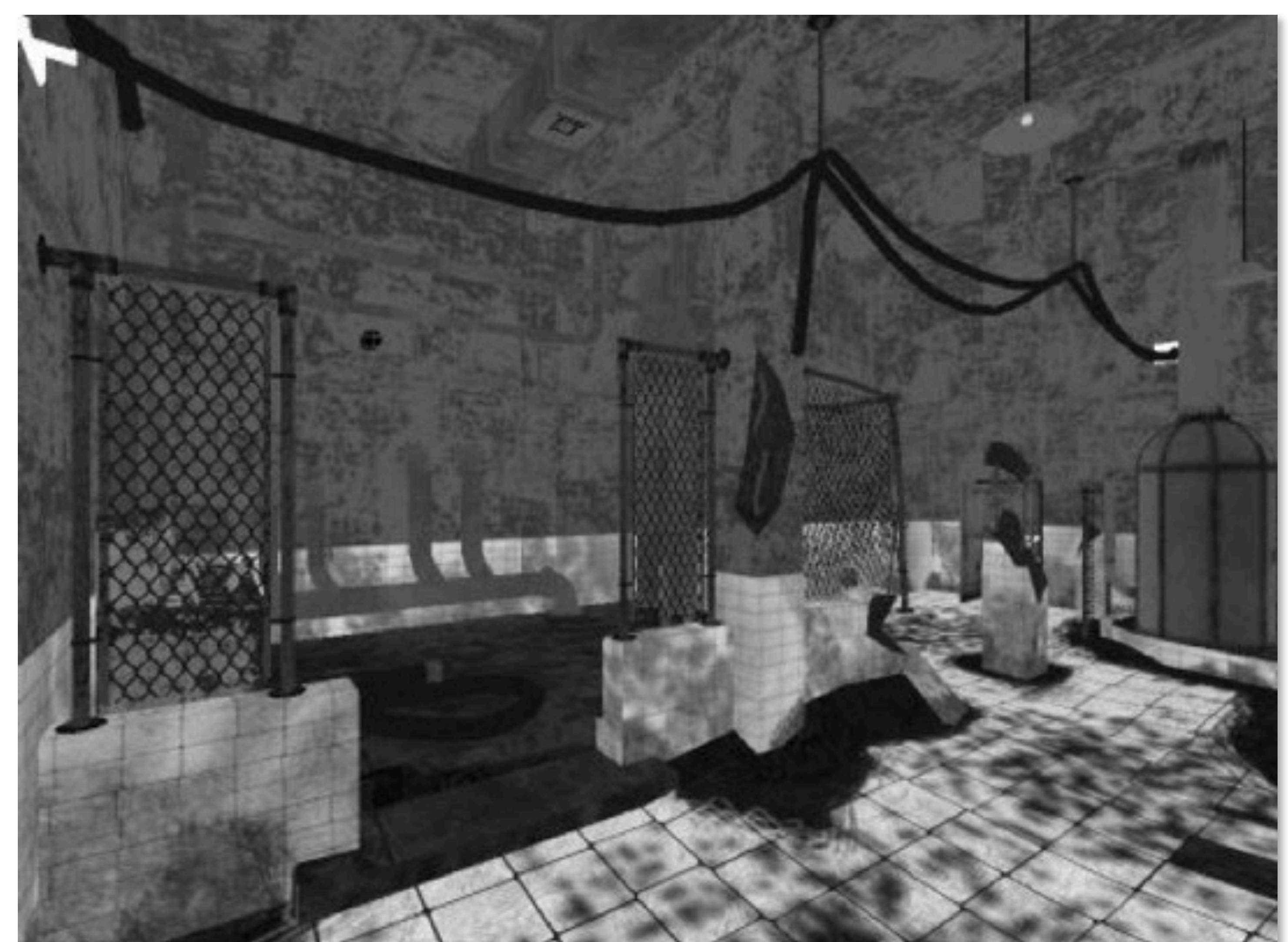
G-buffer = geometry buffer

Albedo (Reflectance)

Depth



Normal



Specular

Image Credit: J. Klint, “Deferred Rendering in Leadworks Engine”

Example G-buffer layout

Graphics pipeline configured to render to four RGBA output buffers (32-bits per pixel, per buffer)

R8	G8	B8	A8	
Depth 24bpp			Stencil	DS
Lighting Accumulation RGB			Intensity	RT0
Normal X (FP16)		Normal Y (FP16)		RT1
Motion Vectors XY		Spec-Power	Spec-Intensity	RT2
Diffuse Albedo RGB			Sun-Occlusion	RT3

Source: W. Engel, "Light-Prepass Renderer Mark III" SIGGRAPH 2009 Talks

Terminology:

Graphics pipeline bound to "multiple render targets"

If G-buffer considered as one big buffer, often referred to as having "fat" pixels

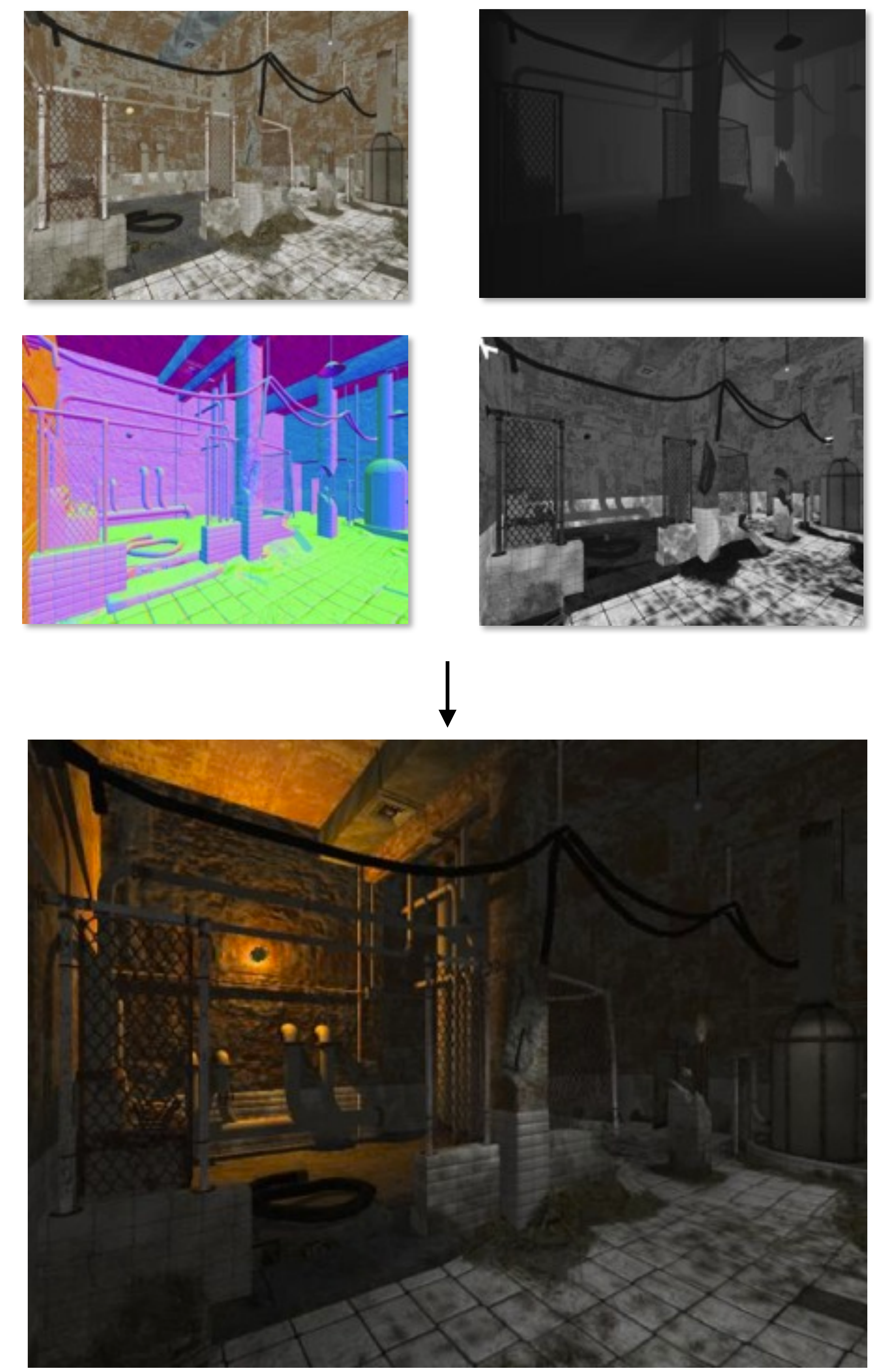
Two-pass deferred shading algorithm

- **Pass 1: geometry pass**
  - Write visible geometry information to G-buffer

- **Pass 2: shading pass**

  For each G-buffer sample, compute shading
  - Read G-buffer data for current sample
  - Accumulate contribution of all lights
  - Output final surface color

Note: Deferred shading produces same result as forward rendering approach, but order of computation is different.



Final Image

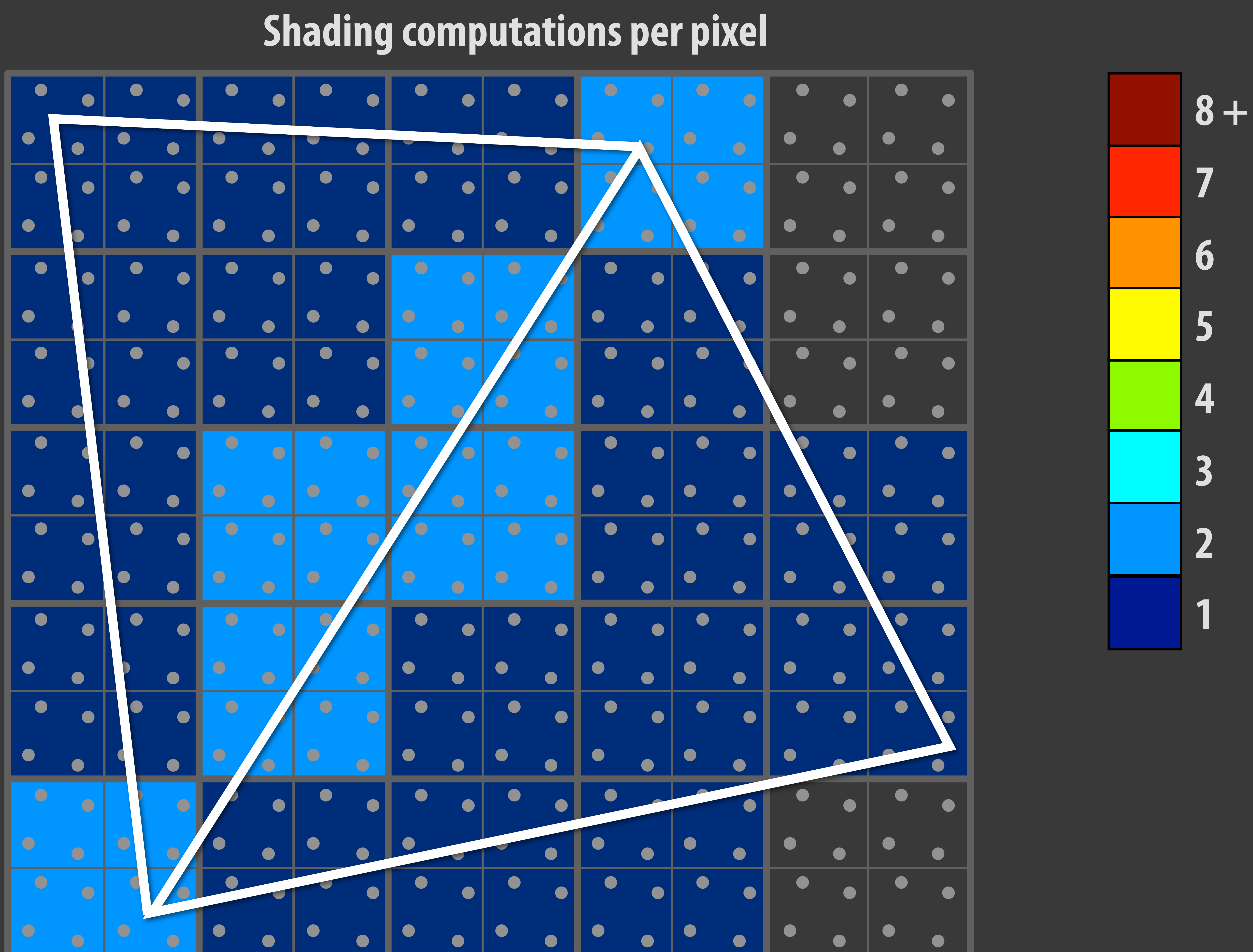
Image Credit: J. Klint, "Deferred Rendering in Leadworks Engine"

Motivation: why deferred shading?

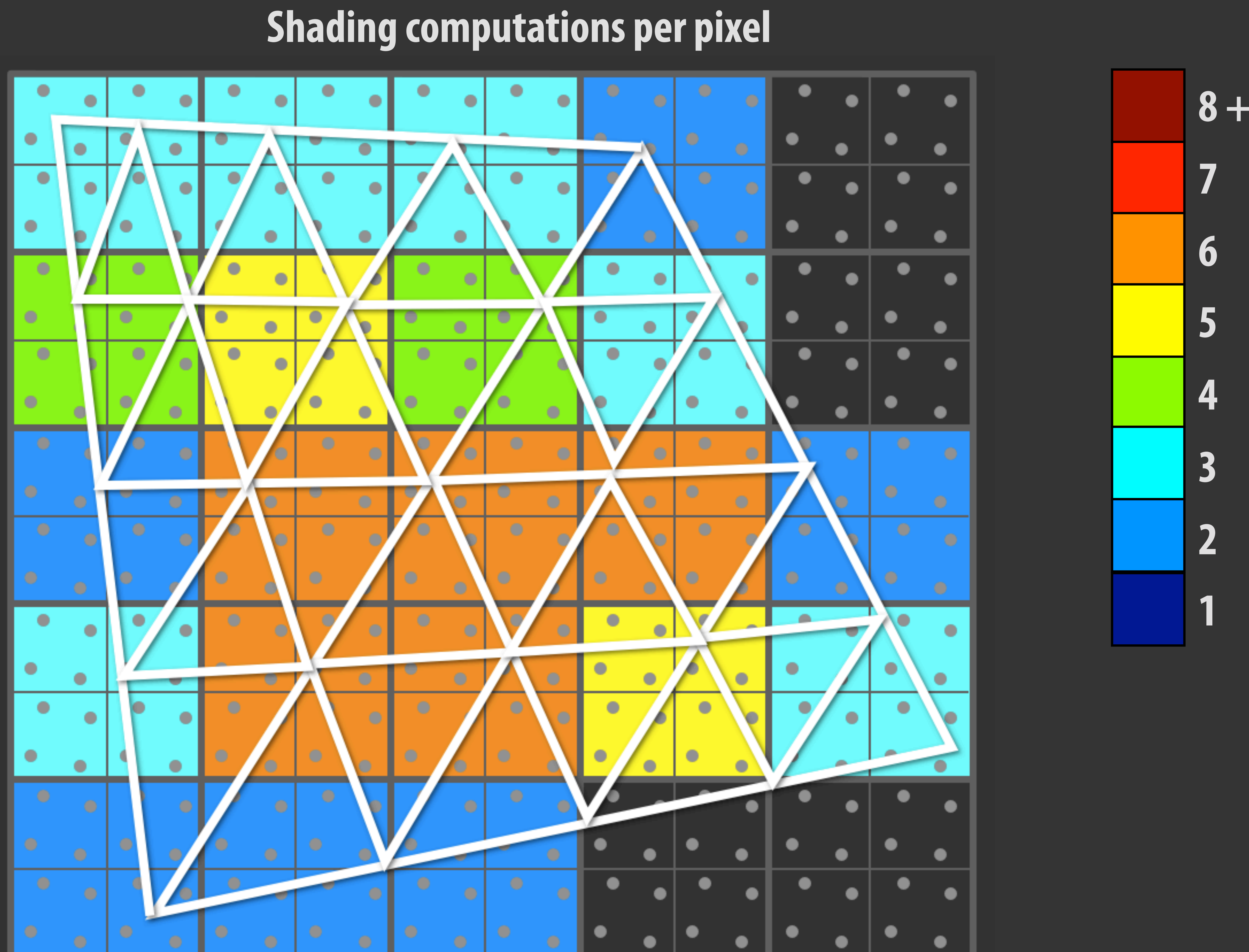
- Shade only surface fragments that are visible
  - Same effect as perfect early occlusion culling
  - But triangle order invariant

- Forward rendering is inefficient when shading small triangles
  - Recall quad-fragment shading granularity: multiple fragments generated for pixels along triangle edges

Recall: forward shading shades multiple fragments at pixels containing triangle boundaries



Recall: forward shading shades multiple fragments at pixels containing triangle boundaries



Motivation: why deferred shading?

- Shade only surface fragments that are visible

- Forward rendering is inefficient when shading small triangles (quad-fragment granularity)

- Increasing complexity of lighting computations
  - Growing interest in scaling scenes to hundreds of light source

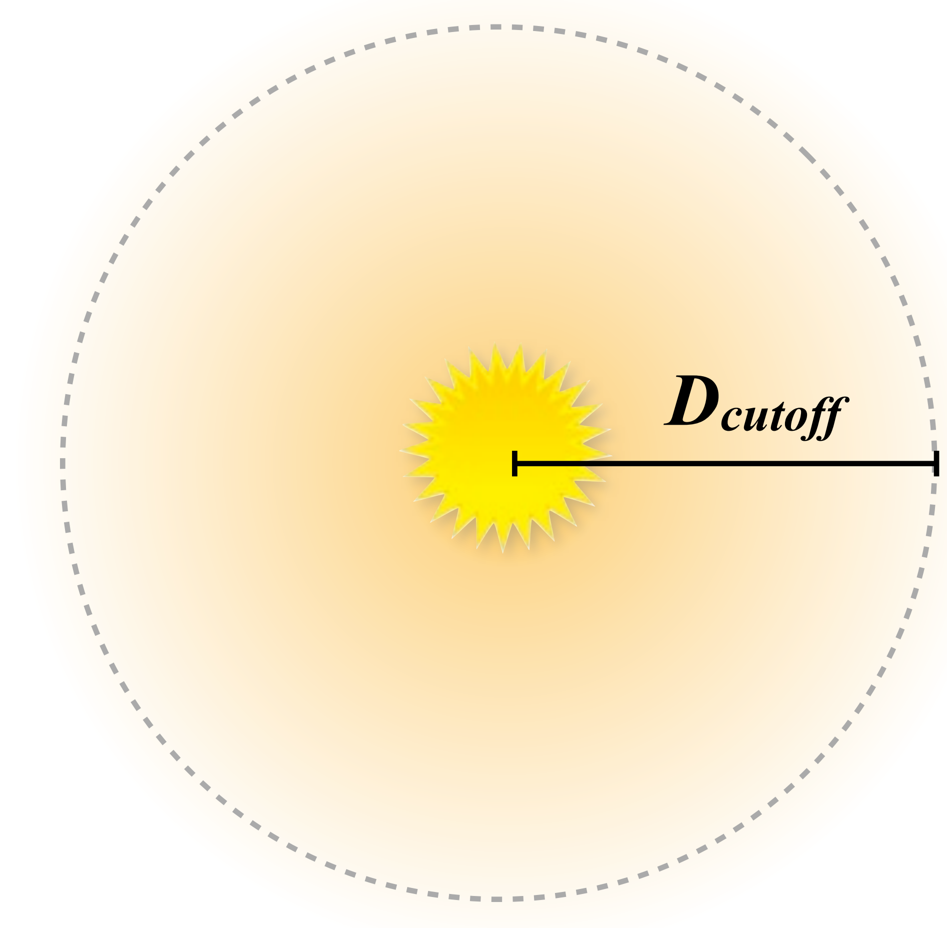
1000 lights

[J. Andersson, SIGGRAPH 2009 Beyond Programmable shading course talk]

Lights

Many different kinds of lights

For efficiency, lights often specify finite volume of influence

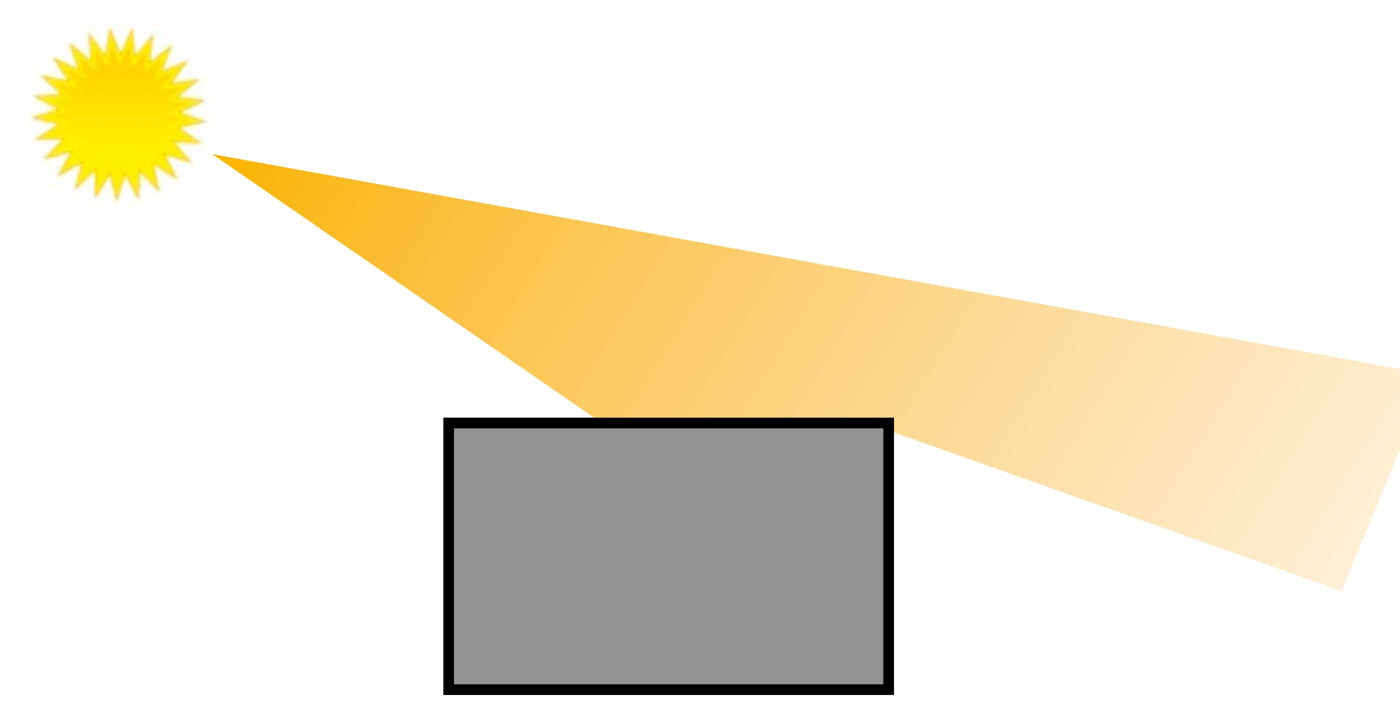


Omnidirectional point light (with distance cutoff)

Directional spotlight

Environment light

Shadowed light



Forward rendering: many-light shader (naive)

```
struct LightDefinition {
  int type;
  ...
}

sampler mySamp;
Texture2D<float3> myTex;
Texture2D<float> myEnvMaps[MAX_NUM_LIGHTS];
Texture2D<float> myShadowMaps[MAX_NUM_LIGHTS];
LightDefinition lightList[MAX_NUM_LIGHTS];
int numLights;

float4 shader(float3 norm, float2 uv)
{
  float3 kd = myTex.Sample(mySamp, uv);
  float4 result = float4(0, 0, 0, 0);
  for (int i=0; i<numLights; i++)
  {
      if (this fragment is illuminated by current light)
      {
         result += // contribution of light to surface reflectance
      }
  }
   return result;
}
```

Large footprint:

Assets for all lights (shadow maps, environment maps, etc.) must be allocated, initialized, and bound to pipeline

Execution divergence:

1. Different outcomes for "is illuminated" test
2. Different logic to perform test (based on light type)
3. Different logic in loop body (based on light type, shadowed/unshadowed, etc.)

Work inefficient:

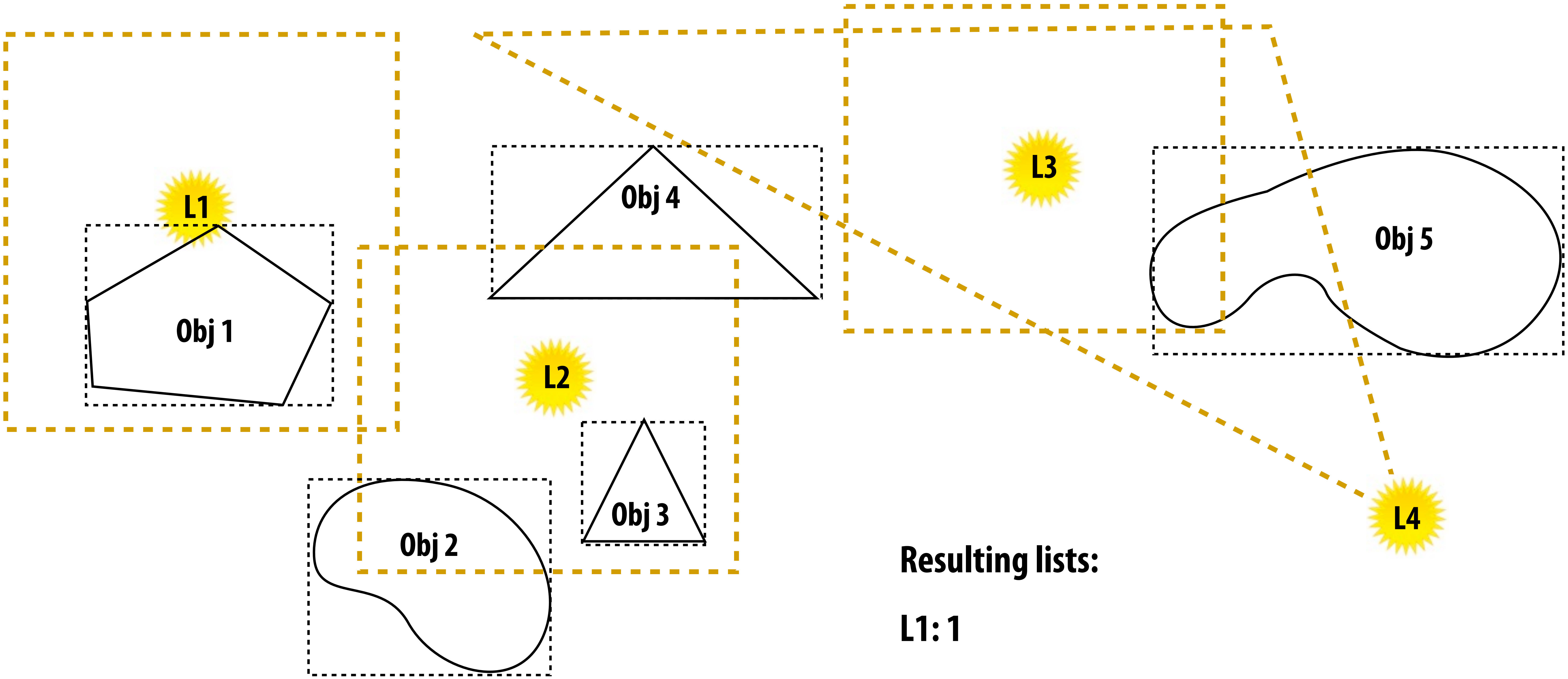
Predicate evaluated for each fragment/light pair
(spatial coherence should exist)

Forward rendering: techniques for scaling to many lights

- **Application maintains light lists**
  - Lights store lists of objects they illuminate
  - CPU builds list by intersecting light volume with scene geometry
    (note, light-geometry interactions computed per light-object pair, not light-fragment pair)

Light lists

Example: Compute lists based on conservative bounding volumes for lights and scene objects



Resulting lists:

L1: 1

L2: 2, 3, 4

L3: 5

L4: 4, 5

Forward rendering: techniques for scaling to many lights

- **Application maintains light lists**
  - Lights store lists of objects they illuminate
  - CPU builds list by intersecting light volume with scene geometry
    (note, light-geometry interactions computed per light-object pair, not light-fragment pair)

- **Option 1: draw scene in smaller batches**
  - Before drawing each object, only bind data for relevant lights
  - Precompile shader variants for different sets of bound lights (4-light version, 8 light version, etc.)
  - Low execution divergence during fragment shading
  - Many state changes, small draw batch sizes (draw call = single object)

- **Option 2: multi-pass rendering**
  - For each light, render scene with additive blending (only render geometry illuminated by light)
  - Minimal footprint for light data
  - Low execution divergence during fragment shading
  - Severe cost of redundant geometry processing, frame-buffer access, redundant execution of common shading sub-expressions in fragment shader

Many-light deferred shading

```
For each light:
    Generate/bind shadow/environment maps
    Compute light's contribution for each G-buffer sample:
        For each G-buffer sample
            Load G-buffer data
            Evaluate light contribution (may be zero)
            Accumulate contribution into frame-buffer
```

- **Good**
  - **Only process geometry once**
  - **Avoids divergent execution in shader**
  - **Outer loop over lights: avoids light data footprint issues**
- **Bad?**

Many-light deferred shading

```
For each light:
    Generate/bind shadow/environment maps
    Compute light's contribution for each G-buffer sample:
        For each G-buffer sample
            Load G-buffer data
            Evaluate light contribution (may be zero)
            Accumulate contribution into frame-buffer
```

- **Bad ***
  - **Limited shading model (G-buffer defines parameters to shader)**
  - **Does not handle transparency**
  - **"Does contribute" predicate evaluated per light-fragment pair**
  - **High bandwidth cost (reload G-buffer each pass, output to frame-buffer)**

(* Will address one more drawback later)

Reducing deferred shading bandwidth costs

- **Process multiple lights in each accumulation pass**
  - Amortize G-buffer load, frame-buffer write across lighting computations for multiple lights
- **Only perform shading computations for G-buffer samples illuminated by light**
  - E.g., Rasterize light volume, only shade covered G-buffer samples (light-fragment predicate evaluated conservatively by rasterizer)
  - Compute screen-aligned quad covered by light volume, only process samples within quad
  - Many techniques for culling light/G-buffer sample interactions

Visualization of number of lights evaluated per G-buffer sample
(scene contains 1024 lights)

Image Credit: A. Lauritzen

Tile-based deferred shading

[Andersson 09]

- Main idea: Compute lights that influence small G-buffer tile, process tile samples x relevant lights as a group

- Efficient implementation enabled by compute shader (think blocking)
  - Amortizes G-buffer load, frame-buffer write across lights
  - Amortizes light data load across tile samples
  - Amortizes light-sample culling across samples in a tile

Tile-based deferred shading

[Andersson 09]

Each thread group is responsible for shading a 16x16 sample tile of the G-buffer

```
LightDescription tileLightList[MAX_LIGHTS];  // group shared memory

Compute Z-min, Zmax for current tile

barrier;

for each light:  // parallelizes across threads in group
   if (light volume intersects tile frustum)
      append to tileLightList // stored in shared memory

barrier;

for each sample:  // parallelizes across threads in group
   result = float4(0,0,0,0)
   load G-buffer data for sample
   for each light in tileLightList:  // no divergence
       result += contribution of light // thread-local data

   store result to appropriate position in frame buffer
```

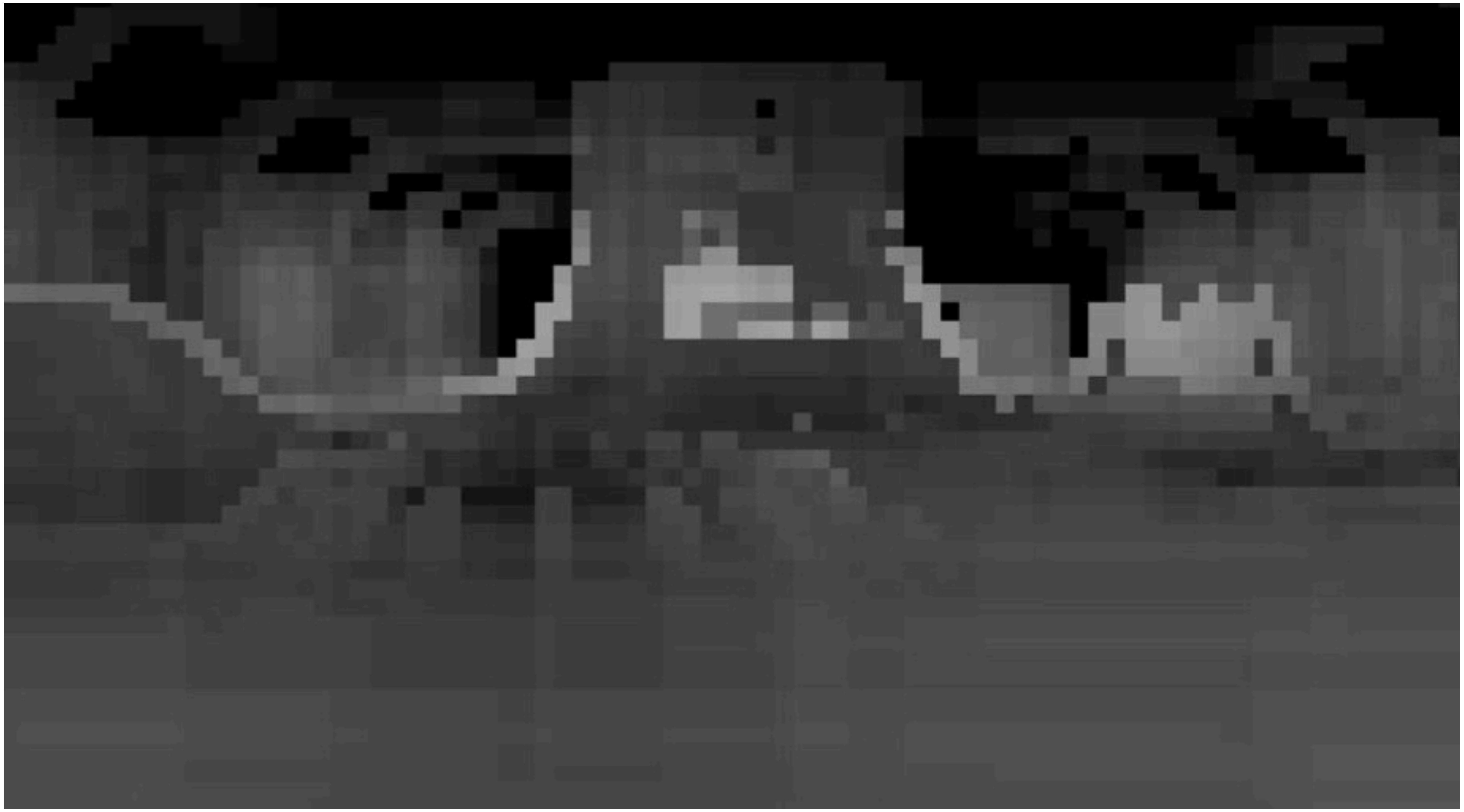
Load depth buffer once

Cull lights at tile granularity

Read G-buffer once

Write to frame buffer once

Tile-based deferred shading: good light culling efficiency

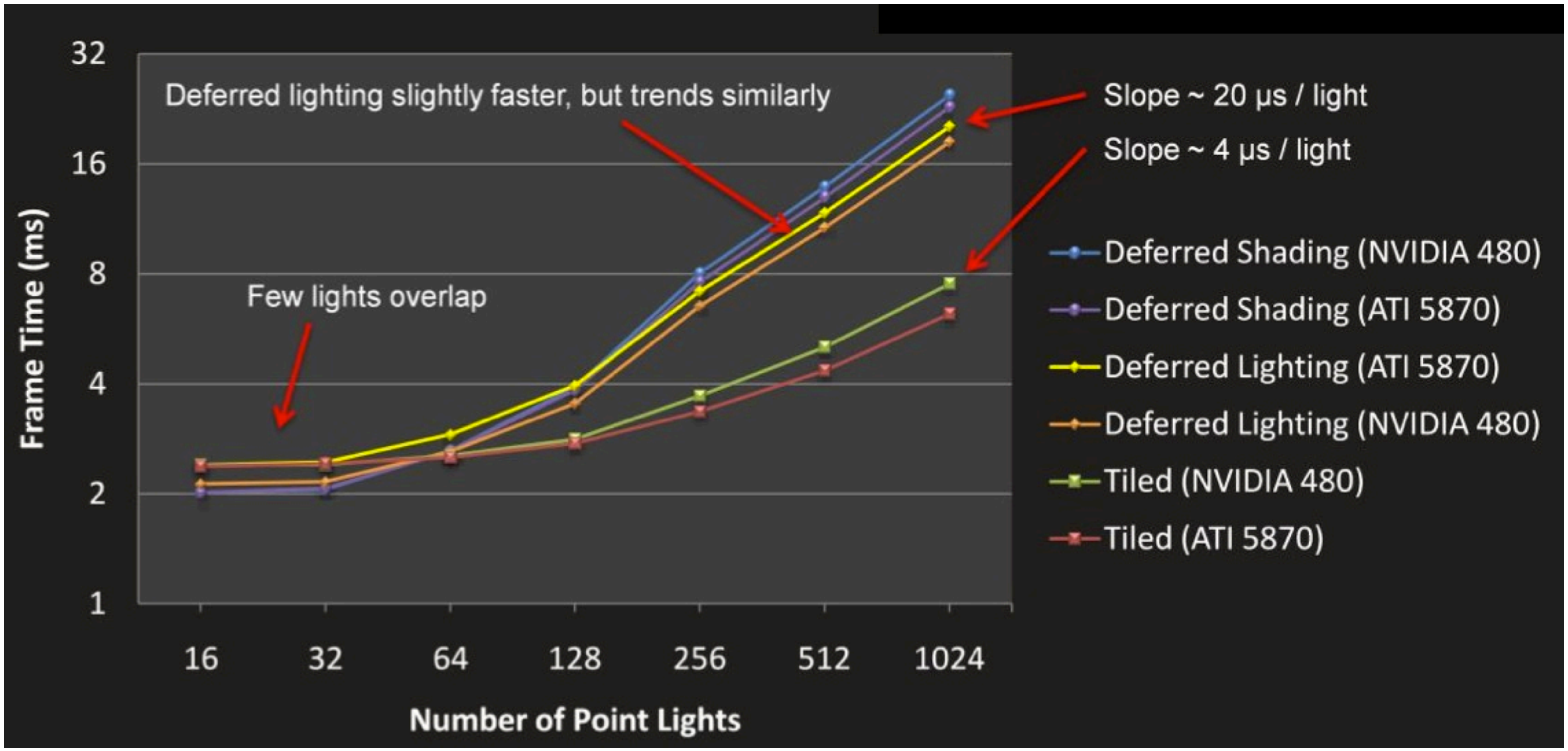


Number of lights evaluated per G-buffer sample
(scene contains 1024 lights)

Image Credit: A. Lauritzen

Tiled vs. conventional deferred shading

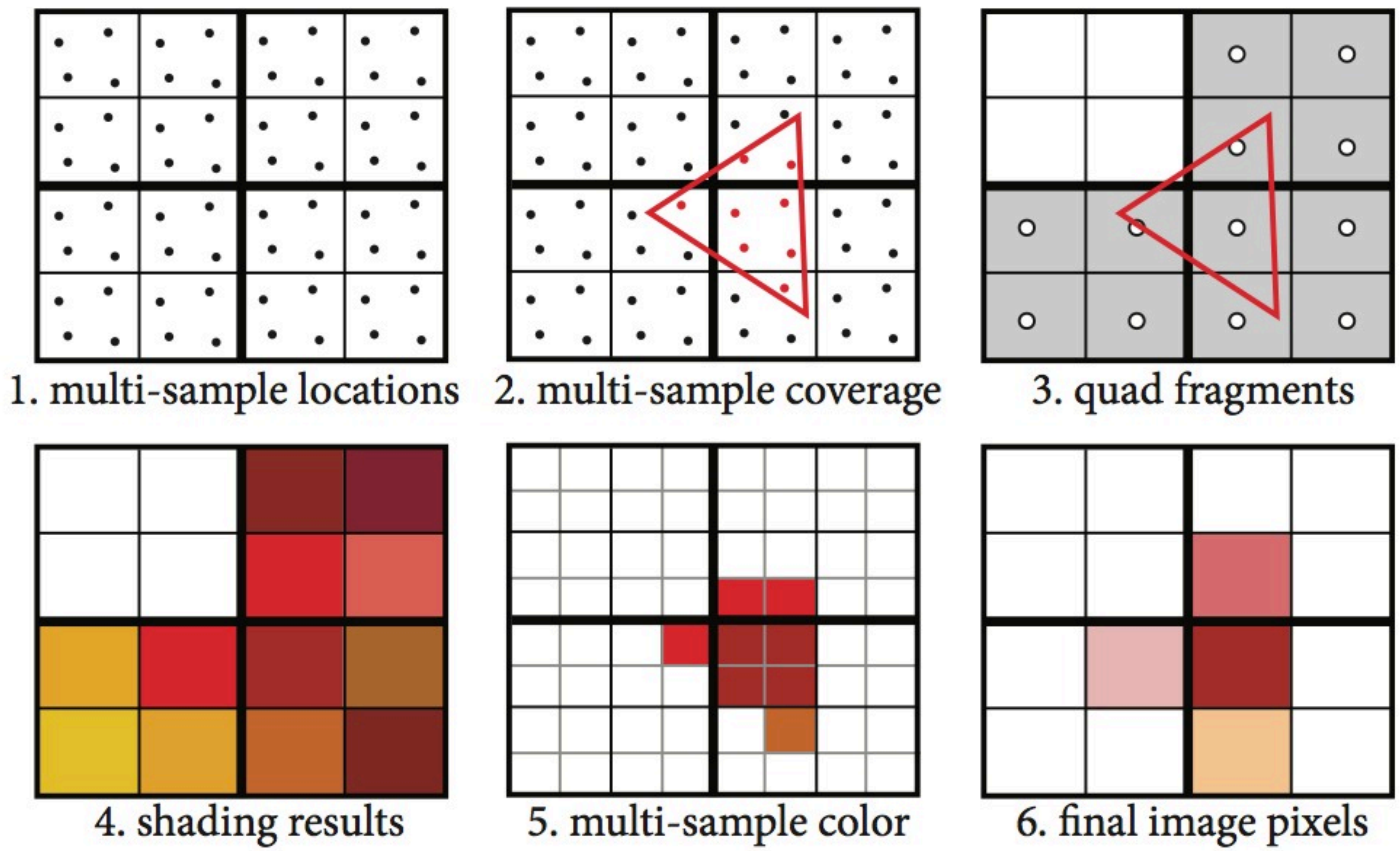
Deferred shading rendering performance: 1920x1080 resolution



[Lauritzen 2009]

Quiz: recall multi-sample anti-aliasing (MSAA)?

Review: MSAA



Main idea: decouple shading sampling rate from visibility sampling rate

- **Depth buffer: stores depth <u>per sample</u>**
- **Color buffer: stores color <u>per sample</u>**
- **Resample color buffer to get final image pixel values**

MSAA in a deferred shading system

- **Challenge: deferred shading is designed to shade exactly once per G-buffer sample**

- **MSAA: shades once <u>per primitive</u> contributing coverage to pixel**
  - **Large triangle assumption: often results one shading computation per pixel**
  - **But extra shading occurs at pixels along primitive boundaries (extra shading necessary to anti-alias silhouettes)**

- **Note: this is also one of the reasons transparency is challenging in a deferred system**

Anti-aliasing solutions for deferred shading

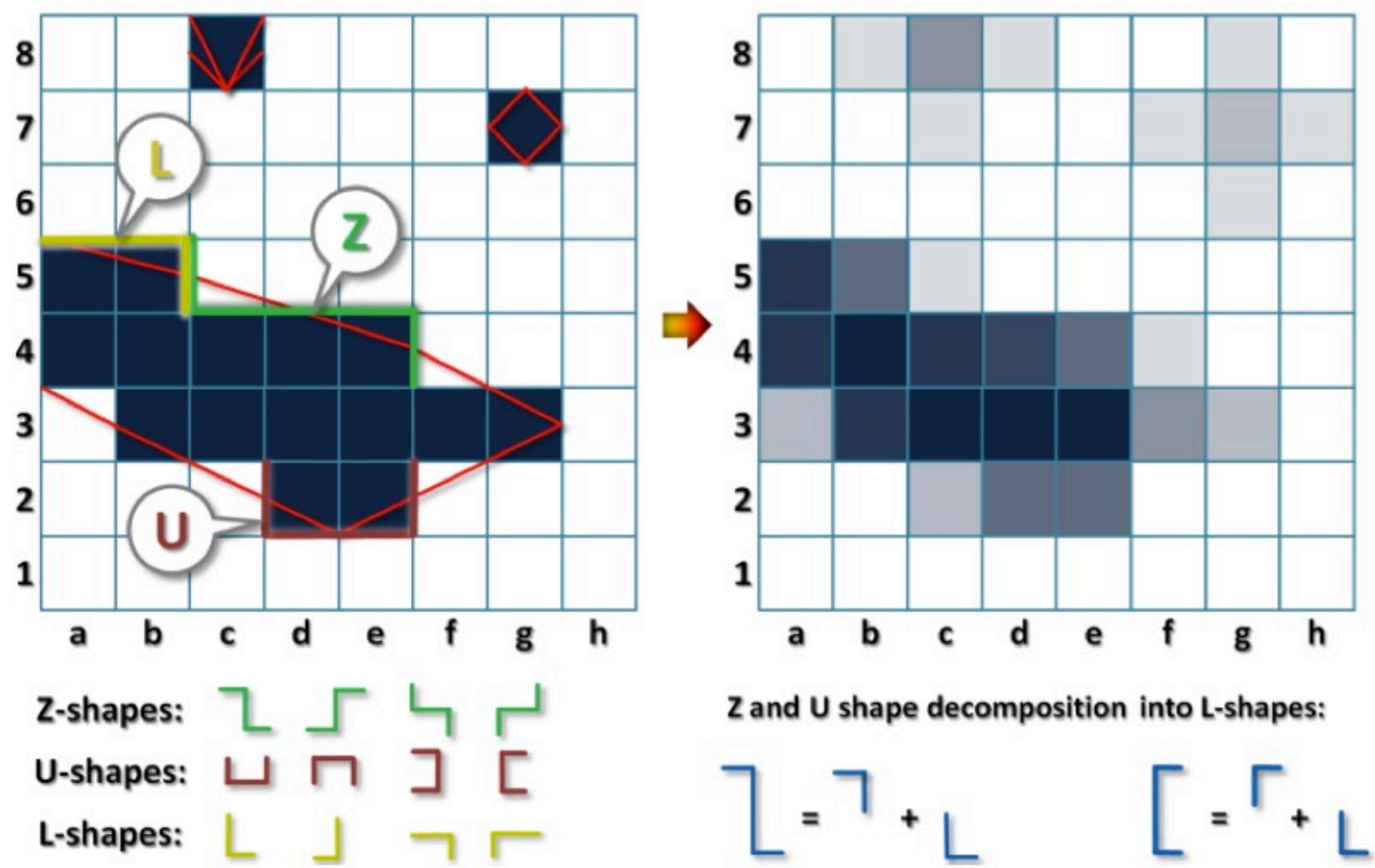
- Super-sample
  - Generate G-buffer larger than frame buffer
  - Shade at G-buffer resolution
  - Downsample result to get final frame-buffer pixels
  - Increases footprint, increases shading cost, increases bandwidth required (but not ratio)

- Intelligently filter frame buffer
  - Identify edges in image and selectively blur frame-buffer near these pixels
  - Same footprint, same shading cost, but produces artifacts
  - Current popular technique: morphological anti-aliasing (MLAA)

Morphological anti-aliasing (MLAA)

[Reshetov 09]

Detect patterns in image
Blend neighboring pixels according to a few simple rules



Morphological anti-aliasing (MLAA)

[Reshetov 09]

Aliased image | Zoomed views (top: aliased, bottom: after MLAA) | After MLAA

Anti-aliasing solutions for deferred shading

- **Super-sample**
  - Increases footprint, increases shading cost, increases bandwidth required (but not ratio)

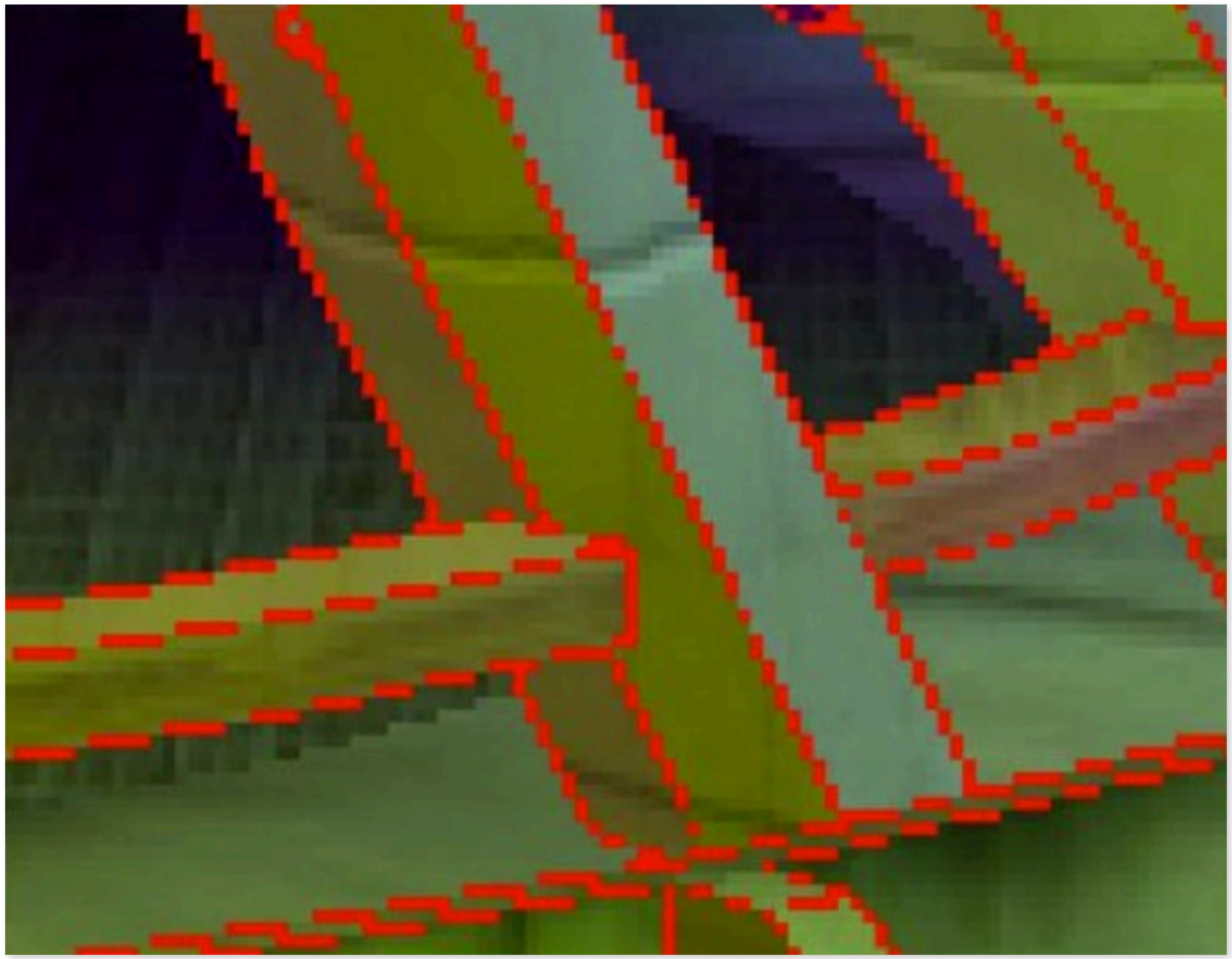
- **Intelligently filter frame buffer (MLAA popular choice)**
  - Same footprint, same shading cost, but produces artifacts

- **Application implements MSAA on its own**
  - Render super-sampled G-buffer
  - Launch one shader instance for each G-buffer pixel, not sample
  - Shader implementation:

```
Detect if pixel contains an edge  // (how is this done robustly?)
If edge:
    Shade all G-buffer samples for pixel (sequentially), combine results
Else:
    Shade one G-buffer sample, store result
```

  - Increased footprint, approx. same shading cost as MSAA, some additional BW cost (to detect edges)

Handling divergence



Red pixels = edges
(Require additional shading)

Increases divergence in shader execution
(recall eliminating shading divergence was one of the motivations of deferred shading)

Can apply standard gamut of data-parallel programming solutions:

Multi-pass:
- pass 1: categorize pixels, set stencil buffer
- pass 2: shade pixels requiring 1 shading computation
- pass 3: flip stencil, shade pixels requiring N shading computations

Standard bandwidth vs. execution coherence trade-off!
(recall earlier in lecture: same principle applied when sorting geometry draw calls by active lights)

Deferred shading summary

- Main idea: perform shading calculations after all geometry processing (rasterization, occlusions) is complete

- Driving motivation in current/near-future systems is scaling scenes to many lights
  - Also, high geometric complexity (due to tessellation) increases overhead of Z-prepass

- Computes (more-or-less) the same result as forward rendering; reorder key rendering loops to change schedule of computation
  - Key loops: for all lights, for all drawing primitives
  - Different <u>footprint</u> characteristics
    - Trade light data footprint for G-buffer footprint
  - Different <u>bandwidth</u> characteristics
  - Different <u>execution coherence</u> characteristics
    - Traditionally deferred shading has traded bandwidth for increased batch sizes and coherence
    - Tile-based methods improve bandwidth requirements considerably
    - MSAA changes bandwidth, execution coherence equation yet again

- Keep in mind: constrains shading model, not used for transparent surfaces

Final comments

- **Which is better, forward or deferred shading?**
  - Often no free lunch

- **Common tradeoff: bandwidth -- execution coherence**
  - Another example of relying on high bandwidth to achieve high ALU utilization
  - In graphics: typically manifest as multi-pass algorithms

- **When considering new techniques, be cognizant of interoperability with existing features and optimizations**
  - Deferred shading not compatible with hardware MSAA implementations (application must role their own)