Lecture 9:

Deferred Shading

Visual Computing Systems
CMU 15-869, Fall 2013

The course so far

The real-time graphics pipeline abstraction

Principle graphics abstractions

Algorithms and modern high performance implementations of those abstractions

Workload characteristics

SPMD Programming abstractions

Shading languages: extending the pipeline with application defined shading functions

General purpose SPMD programming ("compute mode" abstractions)

The GPU processor core implementation and how these abstractions map to these processors

Today... deferred shading

An alternative pipeline structure (and one use of the compute mode abstraction)

We are about to cover several alternative rendering pipelines/algorithms

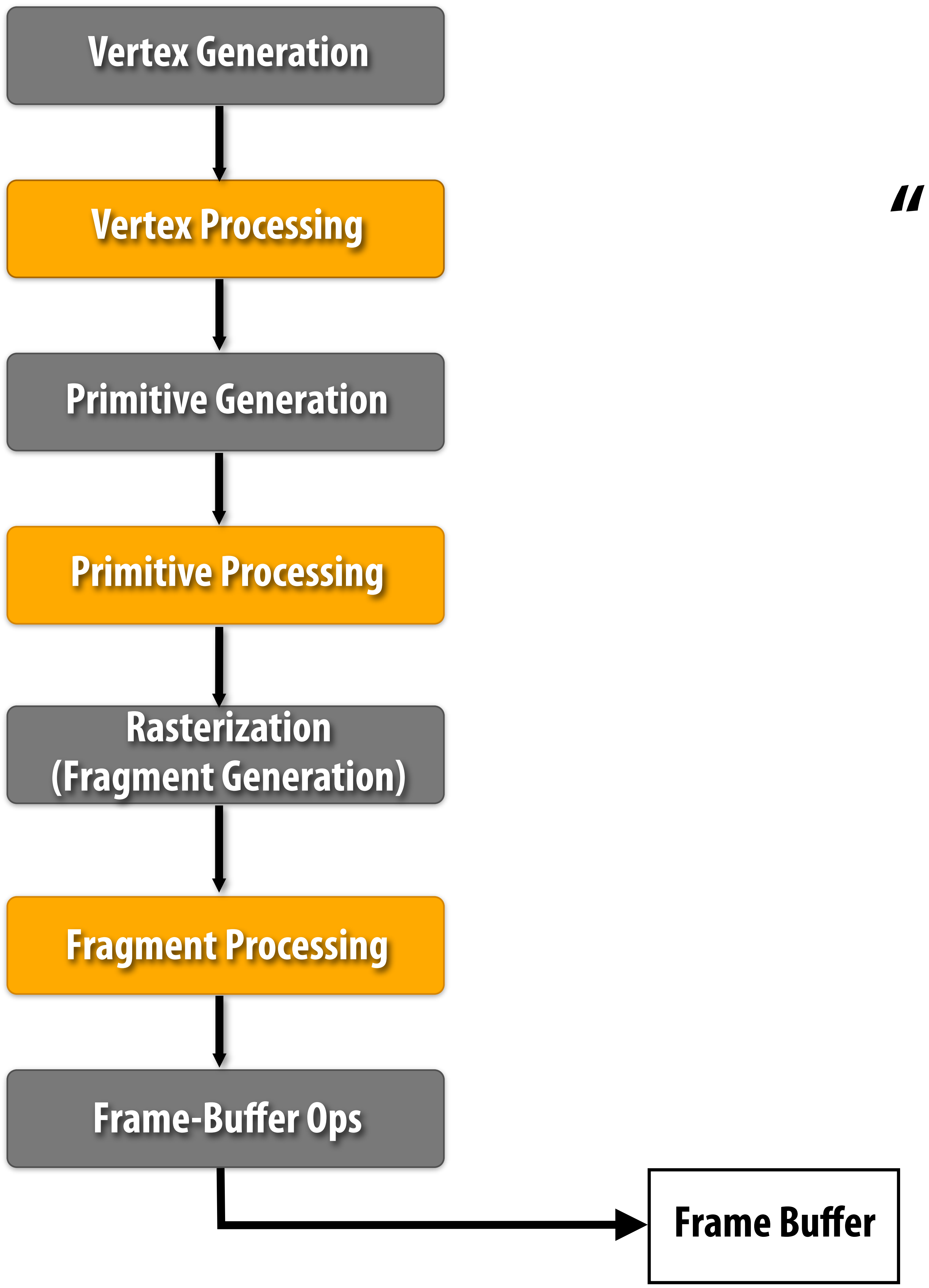
Deferred shading

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

- Not a new idea: implemented in several classic graphics systems, but not directly supported by most high-end GPUs
  - But modern graphics pipeline provides mechanisms to allow application to implement deferred shading efficiently
  - Is natively implemented by mobile GPUs
  - Classic hardware-supported implementations:
    - [Deering et al. 88]
    - UNC PixelFlow [Molnar et al. 92]

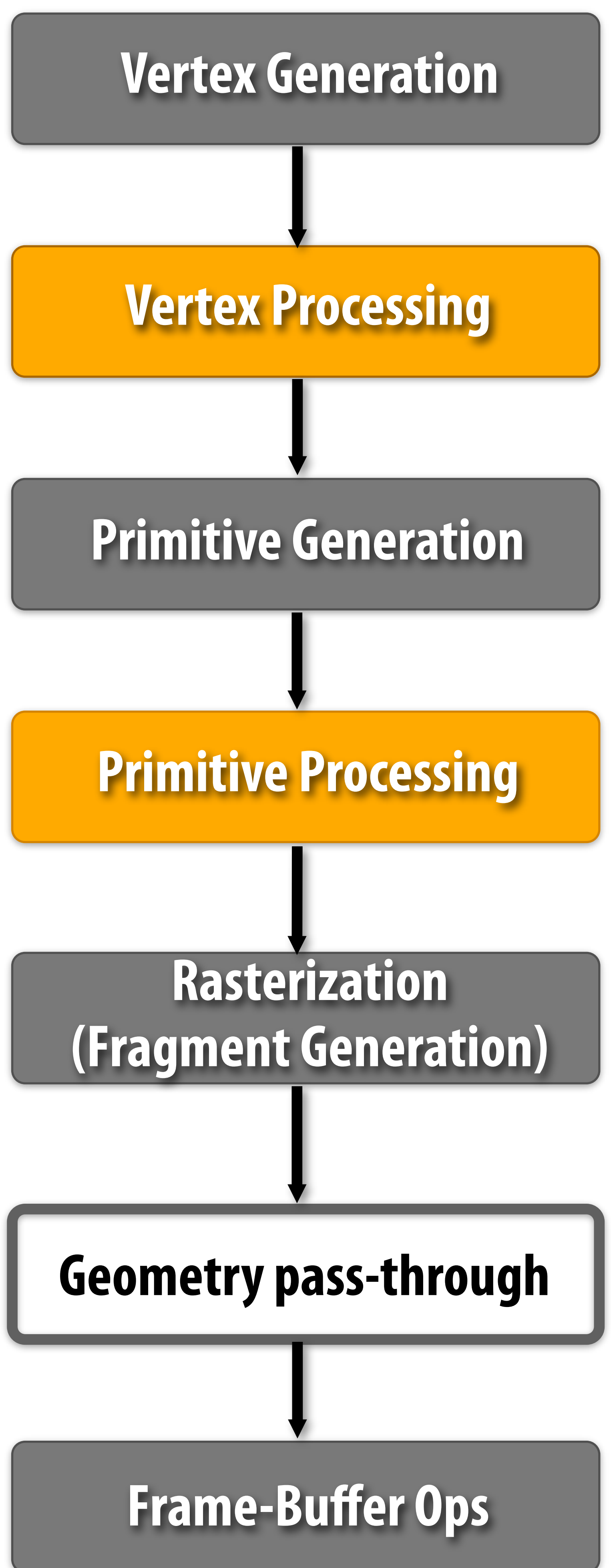
- Popular algorithm for rendering in modern games

The graphics pipeline



"Forward rendering"

Deferred shading pipeline



Two pass approach:

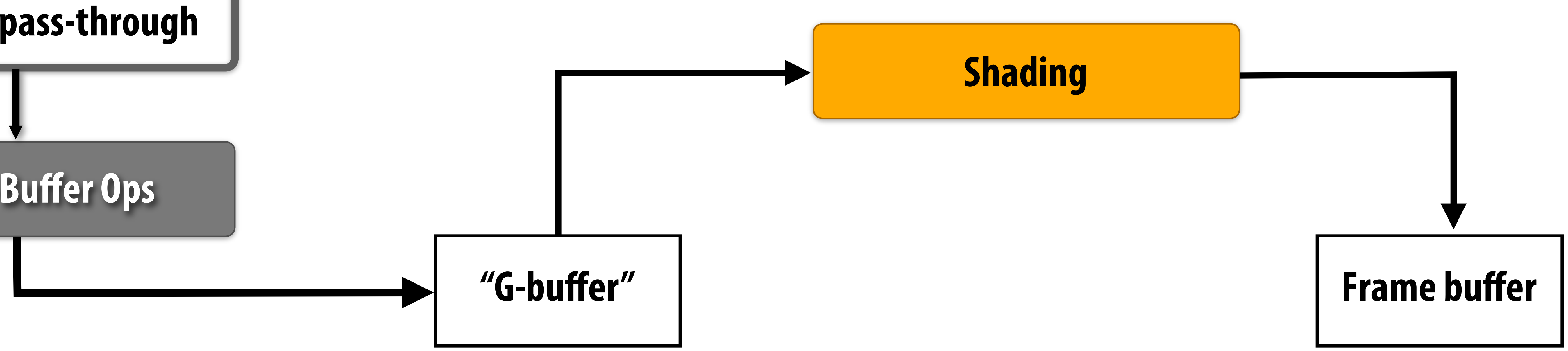
Do not use traditional pipeline to generate RGB image.

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

Rendering output is a screen-size 2D buffer representing information about the surface geometry visible at each pixel
(This buffer is called the "g-buffer", for geometry buffer)

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

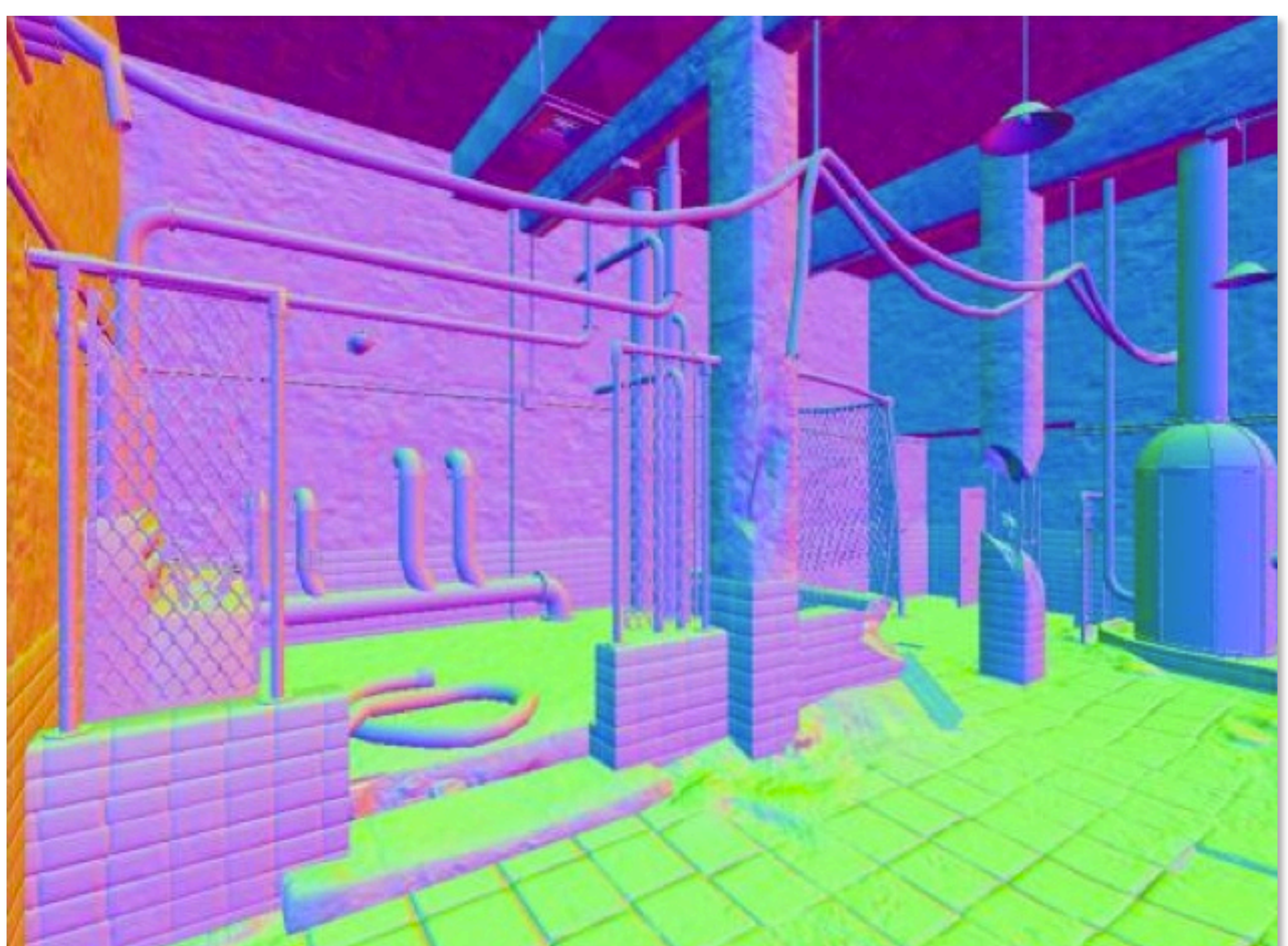
(shading is deferred until all geometry processing -- including all occlusion computations -- is complete)



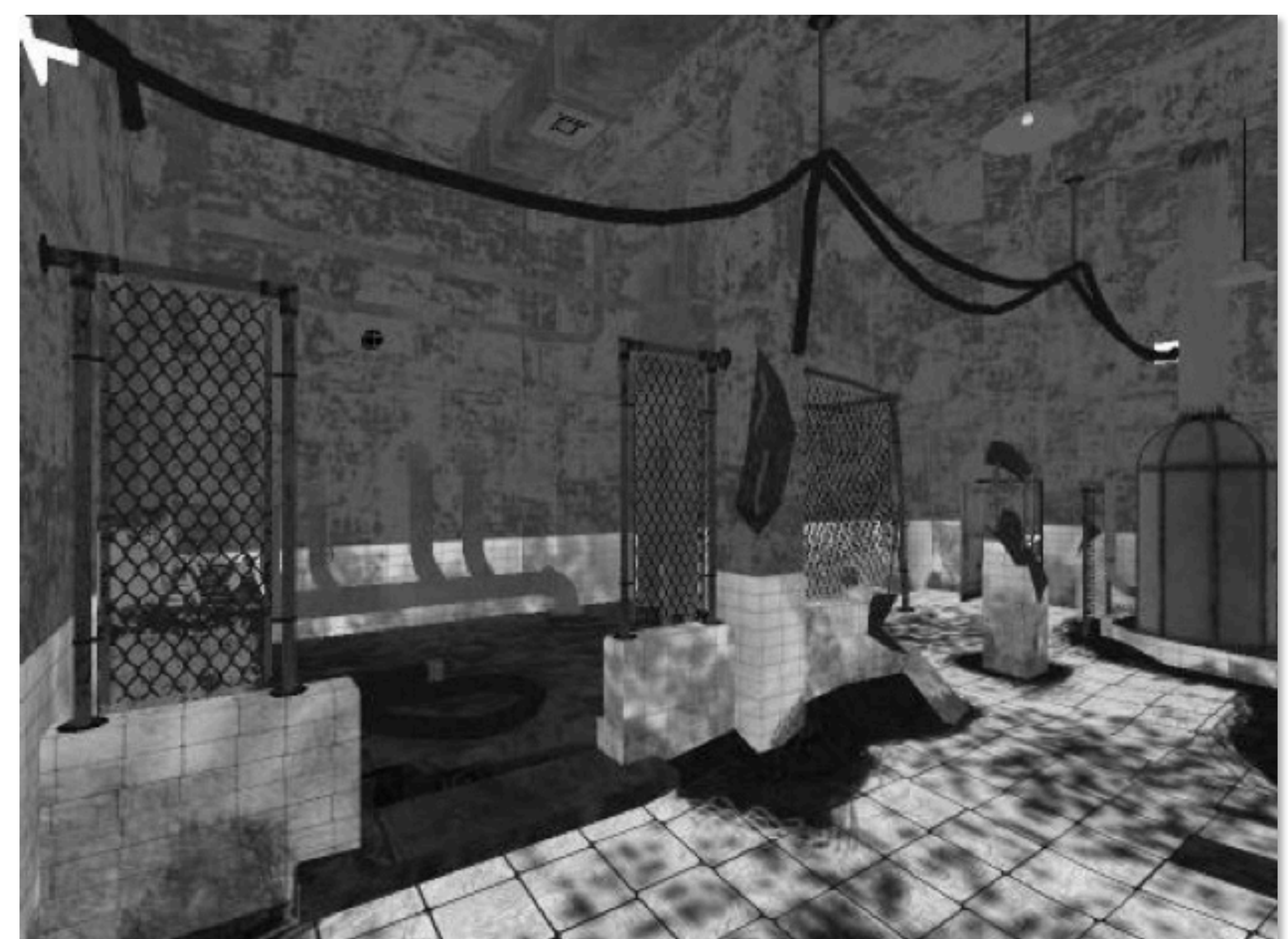
G-buffer = geometry buffer

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

Implementation on modern GPUs:

- Application binds "multiple render targets" (RT0, RT1, RT2, RT3 in figure) to pipeline
- Rendering geometry outputs to depth buffer + multiple color buffers

More intuitive to consider G-buffer as one big buffer with "fat" pixels
In the example above: 32 x 5 = 20 bytes per pixel

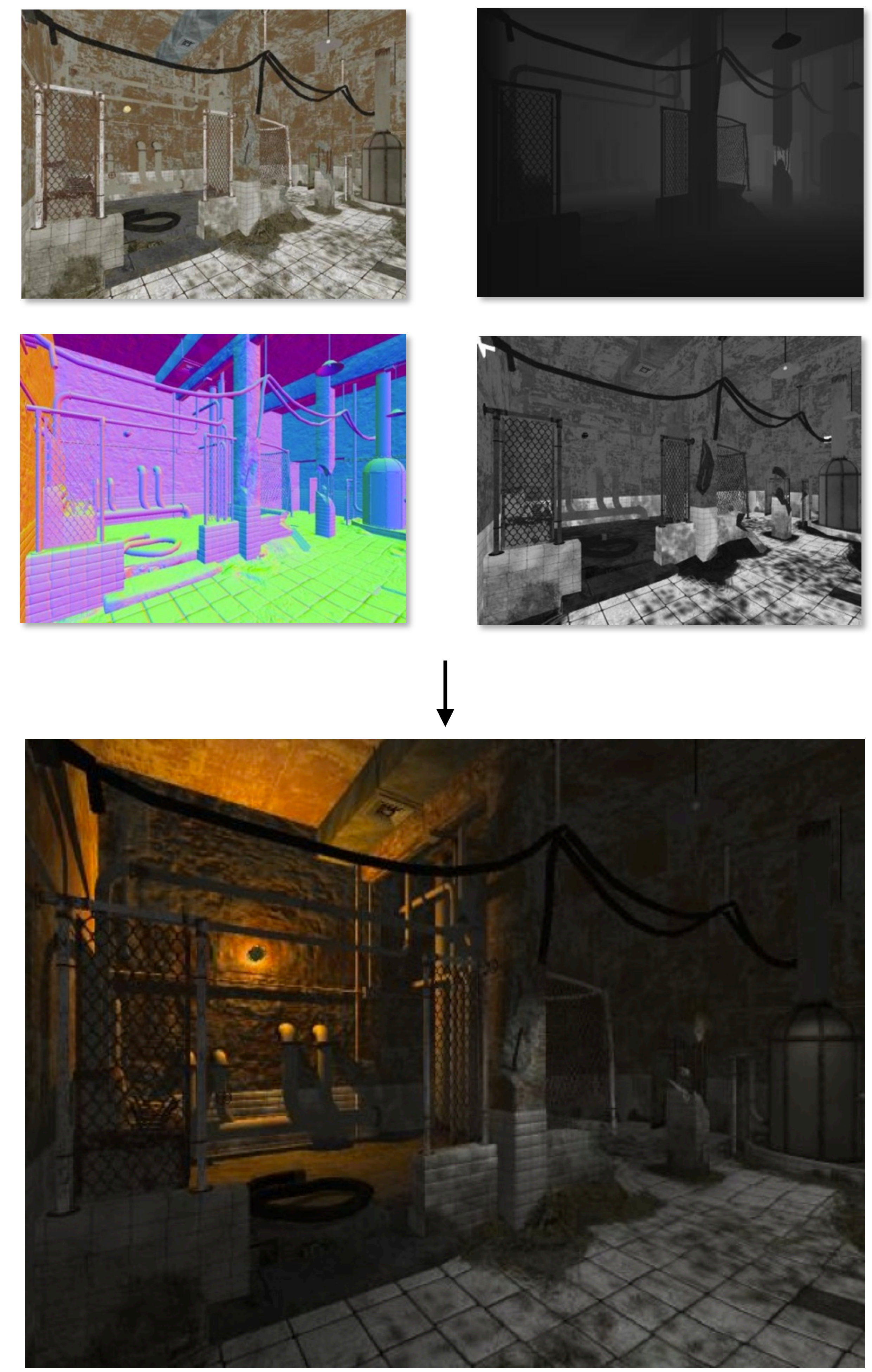
Two-pass deferred shading algorithm

- **Pass 1: geometry pass**
  - Render scene geometry using traditional pipeline
  - 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 for sample

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



Final Image

* Up to order of floating-point operations

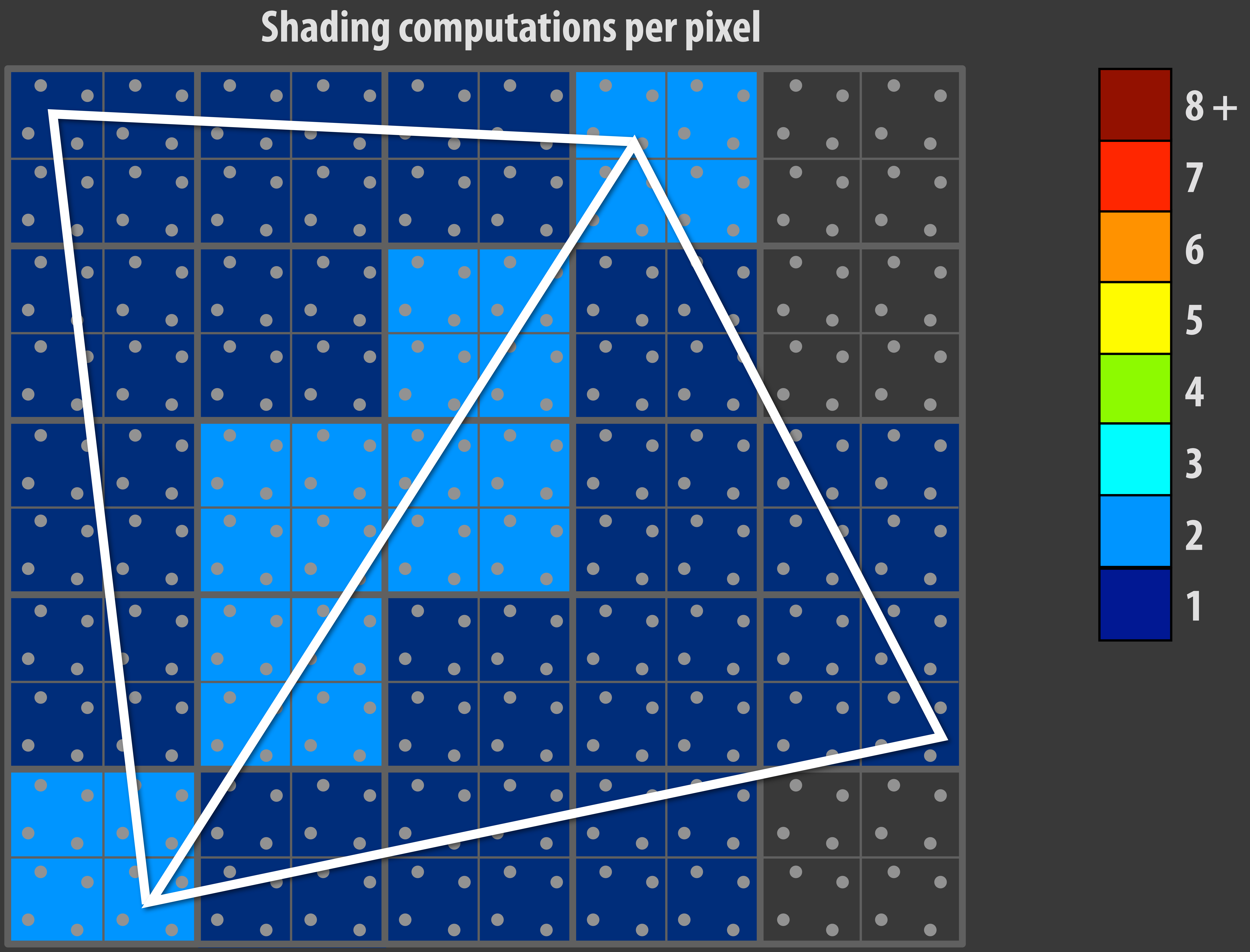
Image Credit: J. Klint, "Deferred Rendering in Leadworks Engine"

Motivation: why deferred shading?

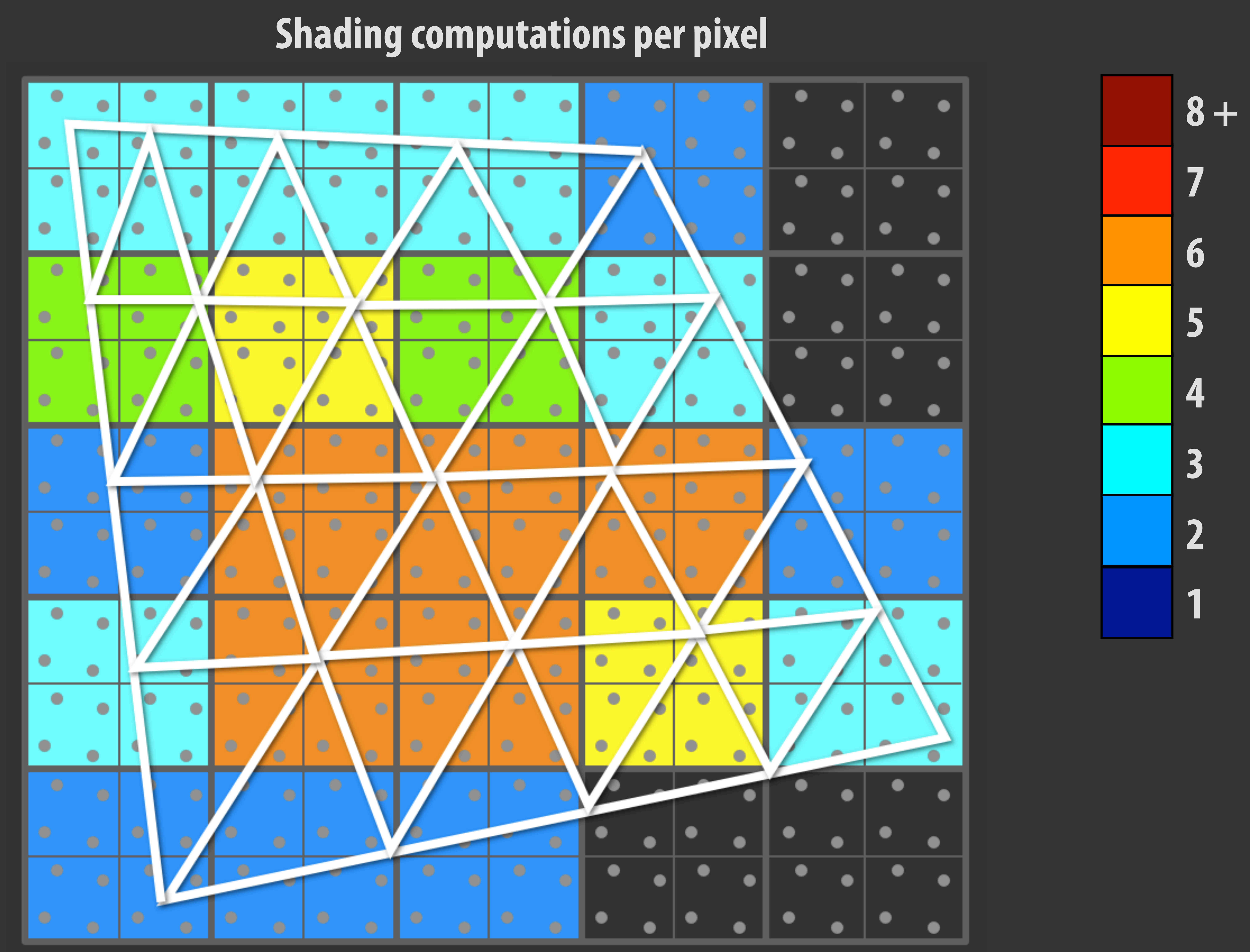
- Shading is expensive: shade only visible fragments
  - Deferred shading has same effect as perfect early occlusion culling
  - But is triangle order invariant (will only shade visible fragments, regardless of application's triangle submission order)

- Forward rendering shades small triangles inefficiently
  - 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 visible surface fragments**

- **Forward rendering shades small triangles inefficiently (quad-fragment granularity)**

- **Increasing complexity of lighting computations**
  - Growing interest in scaling scenes to many light sources

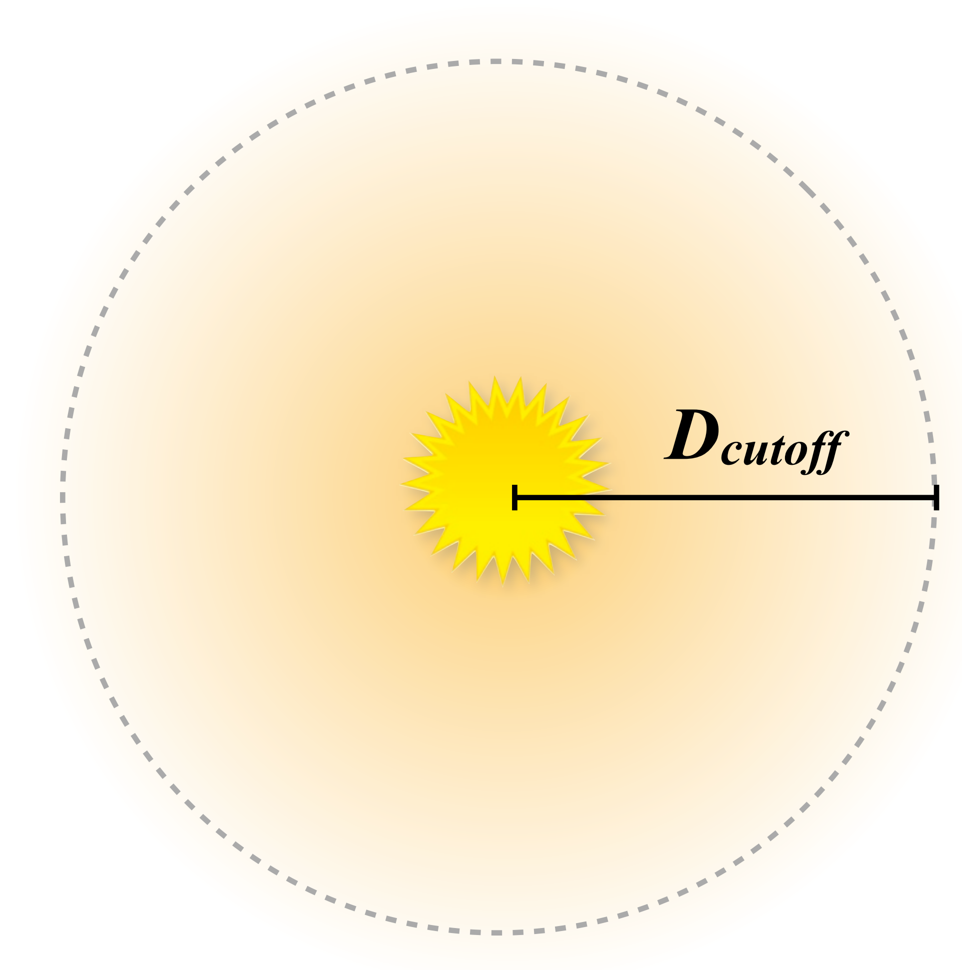
1000 lights

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

Lights

Many different kinds of lights

For efficiency, lights often specify finite volume of influence

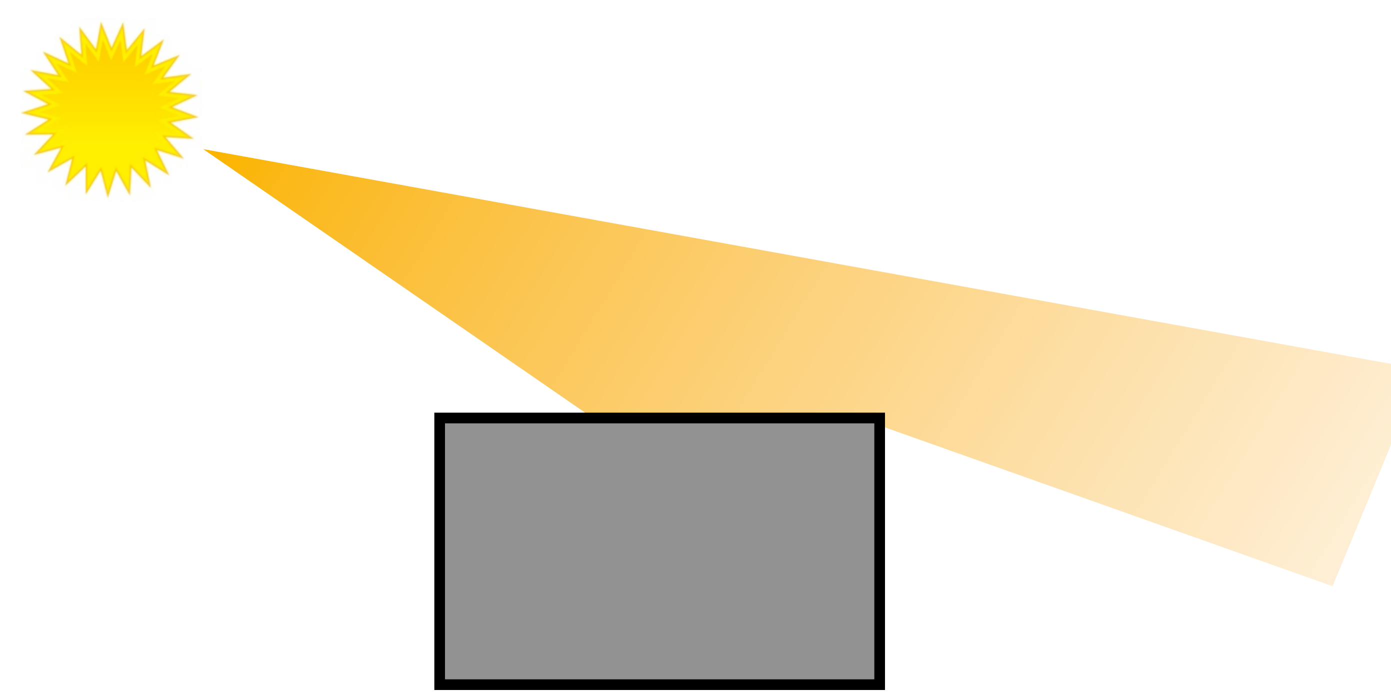


Omnidirectional point light (with distance cutoff)

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 += // eval contribution of light to surface reflectance here
      }
  }
   return result;
}
```

Large footprint:

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

Execution divergence:

1. Different outcomes for "is illuminated" predicate
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:

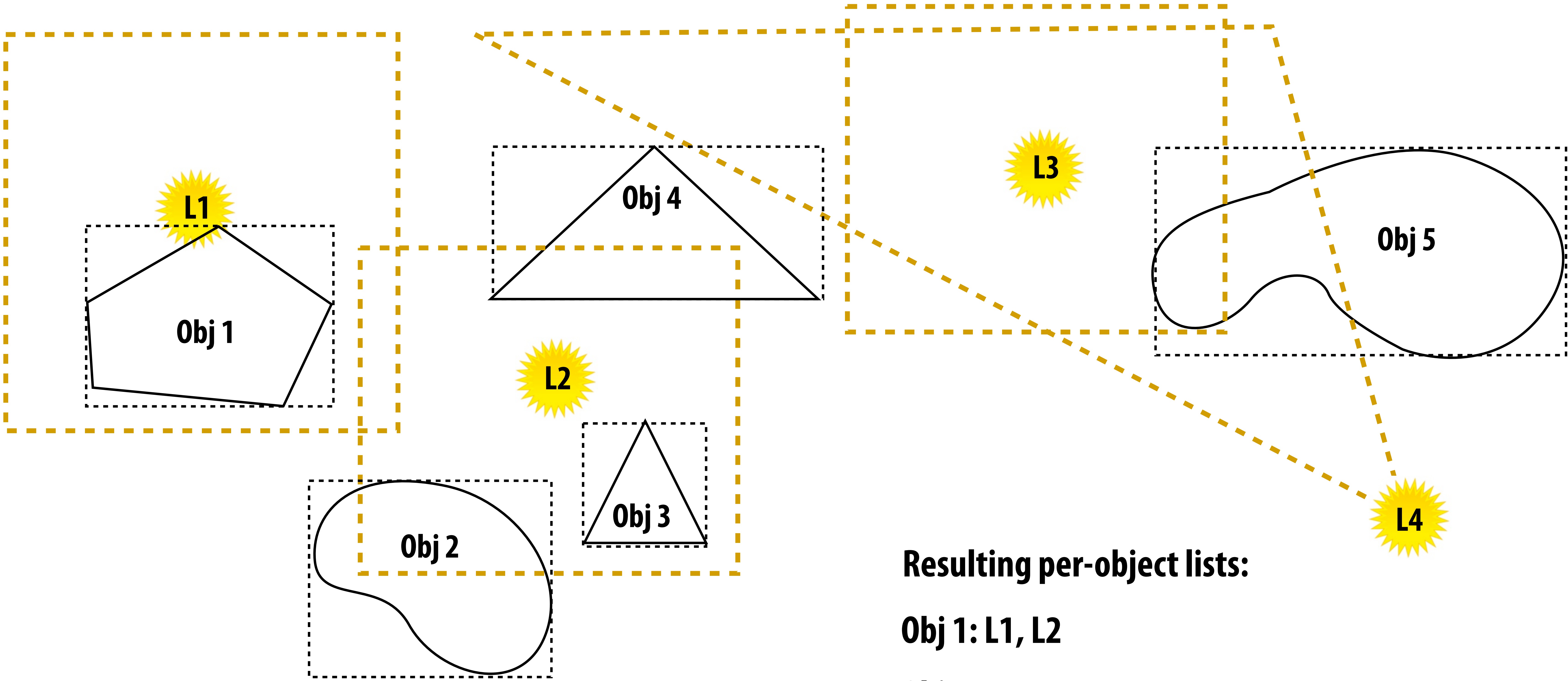
Predicate evaluated for each fragment/light pair:
O(FL) work
  F = number of fragments
  L = nubmer of lights
(spatial coherence in predicate result should exist)

Forward rendering: techniques for scaling to many lights

- **Application maintains light lists**
  - Each object stores lists lights that illuminate it
  - CPU computes list each frame by intersecting light volumes 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 per-object lists:

Obj 1: L1, L2

Obj 2: L2

Obj 3: L2

Obj 4: L2, L4

Obj 5: L3, L4

Forward rendering: techniques for scaling to many lights

- Application maintains light lists
  - Computed conservatively per frame

- Option 1: draw scene in small batches
  - First generate data structures for all lights: e.g., shadow maps
  - 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...)
    - Low execution divergence during fragment shading
    - Many graphics state changes, small draw batch sizes (draw call = single object) *

Stream over scene geometry

- Option 2: multi-pass rendering
  - Compute per-light lists (for each light, compute illuminated objects)
  - For each light:
    - Compute necessary data structures (e.g., shadow maps)
    - Render scene with additive blending (only render geometry illuminated by light)
  - Minimal footprint for light data
  - Low execution divergence during fragment shading
  - Significant overheads: redundant geometry processing, many frame-buffer accesses, redundant execution of common shading sub-expressions in fragment shader

Stream over lights

* Optimized applications will sort geometry by number of lights in list in order to minimize total number of graphics pipeline state changes

Many-light deferred shading

```
Generate G buffer
For each light:
   Generate/bind light's shadow/environment maps
   For each G-buffer sample:  // Compute light's contribution for each G-buffer sample
       Load G-buffer data
       Evaluate light contribution  // may be zero if light doesn't illuminate surface sample
       Accumulate contribution into frame buffer
```

- **Good**
  - **Only process scene geometry once (stream over geometry)**
  - **Avoids divergent execution in shader**
  - **Outer loop is over lights: avoids light data footprint issues (stream over lights)**
  - **Recall other deferred benefits: only shade visibility samples (and no more)**
- **Bad?**

Many-light deferred shading

```
Generate G buffer
For each light:
   Generate/bind light's shadow/environment maps
   For each G-buffer sample:  // Compute light's contribution for each G-buffer sample
       Load G-buffer data
       Evaluate light contribution  // may be zero if light doesn't illuminate surface sample
       Accumulate contribution into frame buffer
```

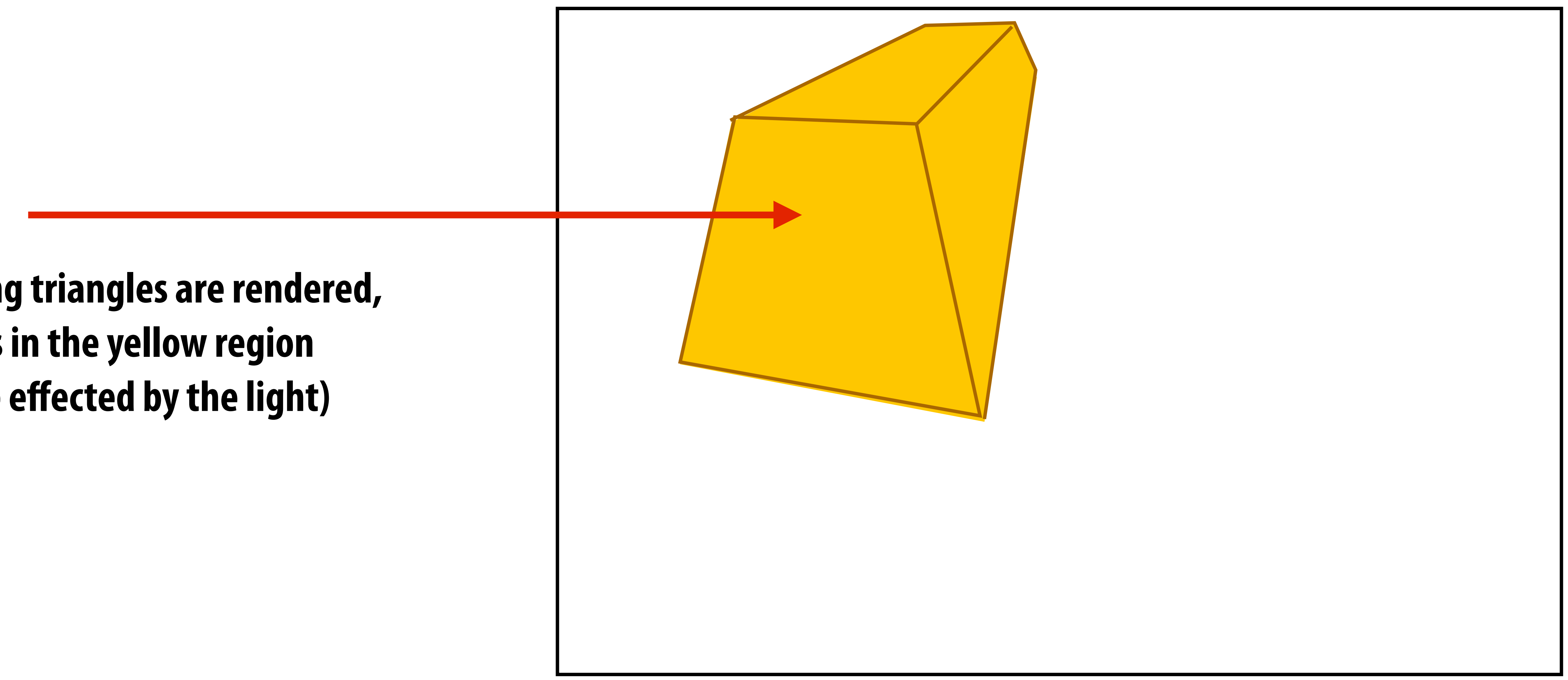
- **Bad**
  - High G-buffer footprint costs: G-buffer has large footprint
    - Especially when G-buffer is supersampled!
  - High bandwidth costs (reload G-buffer each pass, output to frame-buffer)
    - Also, color compression techniques may not work as well for shader input values
  - One shade per frame-buffer sample
    - Does not support transparency (need multiple fragments per pixel)
    - Challenging to implement MSAA efficiently (more on this to come)

Reducing deferred shading bandwidth costs

- **Process multiple lights in each accumulation pass**
  - Amortize G-buffer load and frame-buffer write across lighting computations for multiple lights
- **Only perform shading computations for G-buffer samples illuminated by light**
  - Technique 1: rasterize geometry of light volume, (will only generate fragments for covered G-buffer samples) (light-fragment interaction predicate is evaluated by rasterizer)
  - Technique 2: CPU computes screen-aligned quad covered by light volume, renders quad
  - Many other techniques for culling light/G-buffer sample interactions

Light volume geometry

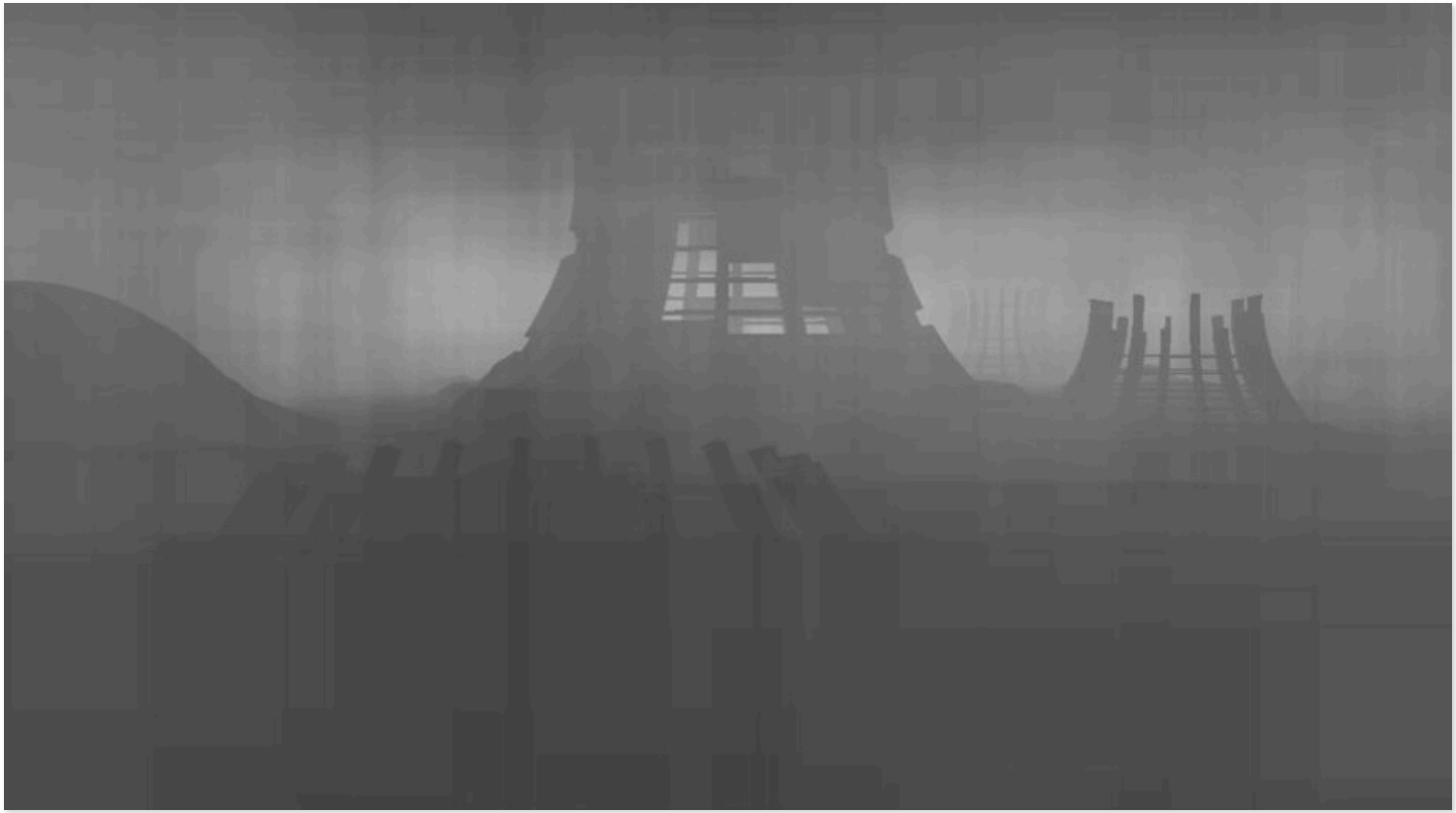
If volume is convex and only front-facing triangles are rendered, rasterizer will only generate fragments in the yellow region (these are the only samples that can be effected by the light)



Visualization of light-sample interaction count

Per-light culling performed using screen-aligned quad per light

(depth of quad is nearest point in light volume: early Z will cull fragments behind scene geometry)



Number of lights evaluated per G-buffer sample

(scene contains 1024 point lights)

Image Credit: A. Lauritzen

Tile-based deferred shading

[Andersson 09]

- Main idea: exploit coherence in light-sample interactions
  - Compute set of lights that influence a small tile of G-buffer samples, then compute contribution of lights to samples in the tile

- Efficient implementation enabled by compute shader
  - Amortize G-buffer load, frame-buffer write across all lights
  - Amortize light data load across tile samples
  - Amortize light-sample culling across samples in a tile

Tile-based deferred shading

[Andersson 09]

Each compute shader thread group is responsible for shading a 16x16 sample tile of the G-buffer (256 threads per group)

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

All threads cooperatively compute Z-min, Zmax for current tile   <-- Load depth buffer once

barrier;

for each light:  // parallel across threads in thread group (parallel over lights)
   if (light volume intersects tile frustum)   <-- Cull lights at tile granularity
      append light to tileLightList // stored in shared memory

barrier;

for each sample:  // parallel across threads in group (parallel over samples)
   result = float4(0,0,0,0)
   load G-buffer data for sample   <-- Read G-buffer once
   for each light in tileLightList:  // no divergence across samples
       result += evaluate contribution of light

   store result to appropriate position in frame buffer   <-- Write to frame buffer once
```

Tiled-based light culling

Yellow boxes: screen-aligned light volume bonding boxes
Blue boxes: screen tile boundaries

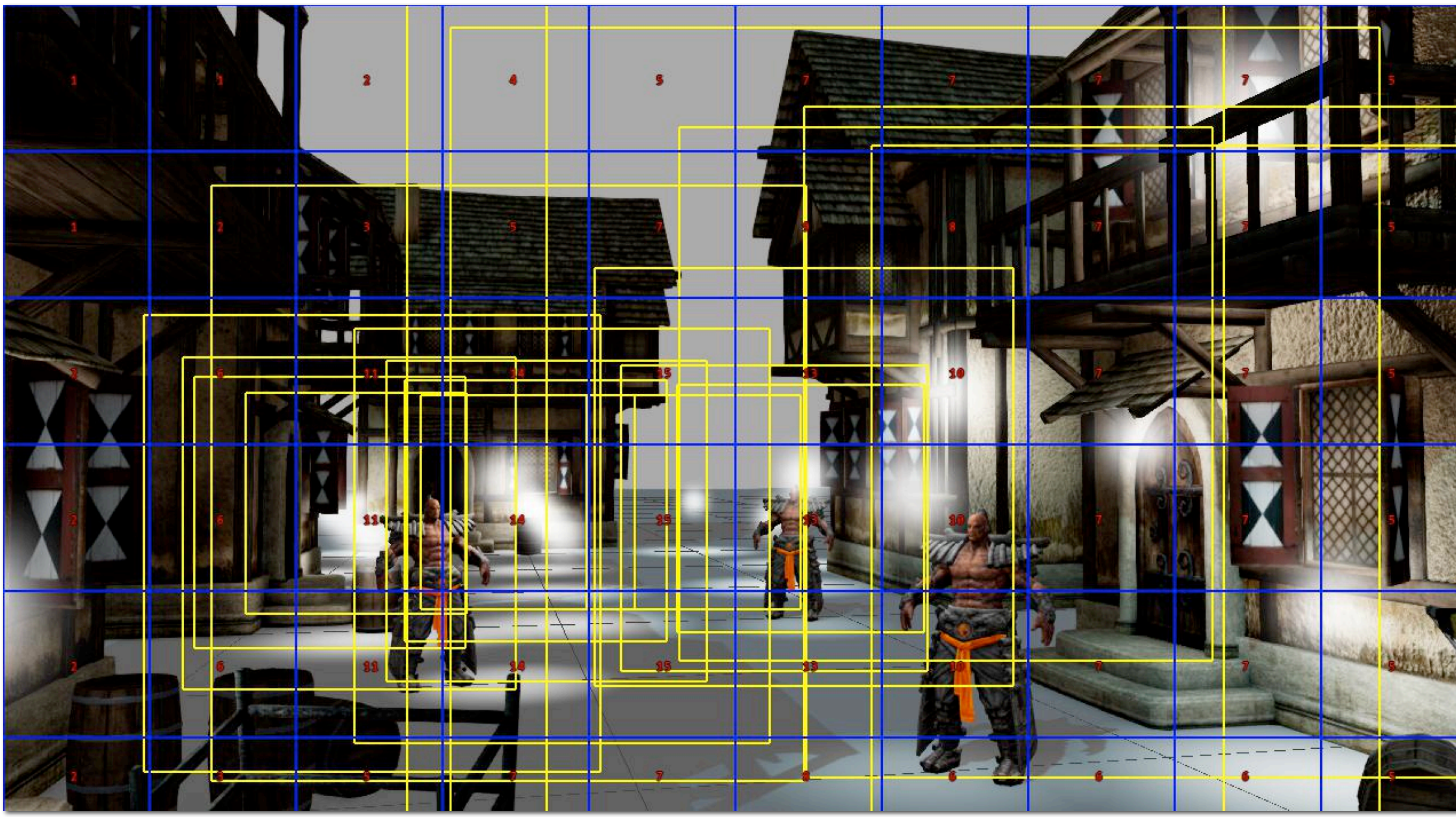
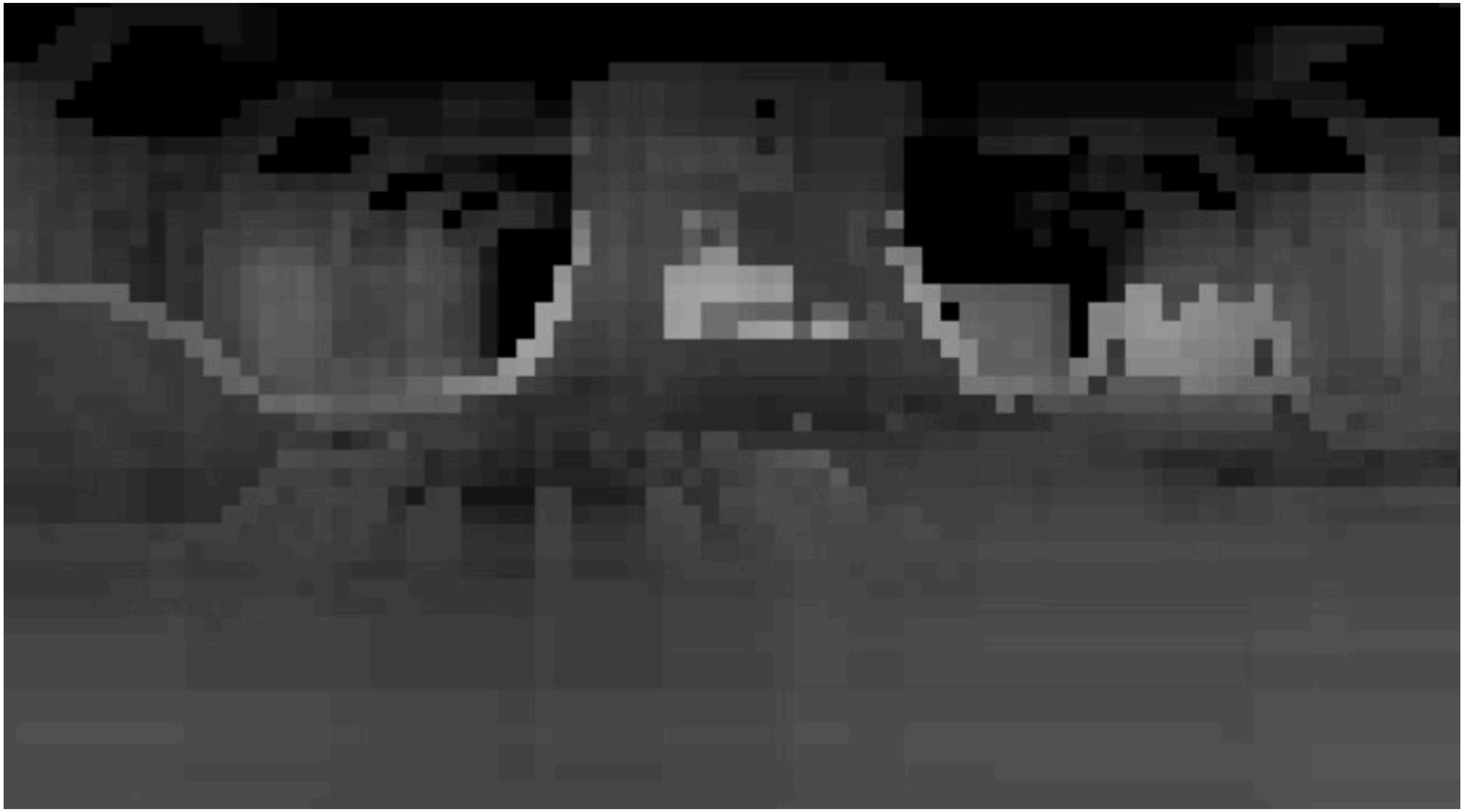


Image credit: HMREngine: http://www.hmrengine.com/blog/?p=399

Tile-based deferred shading: good light culling efficiency

16x16 granularity of light culling is visible

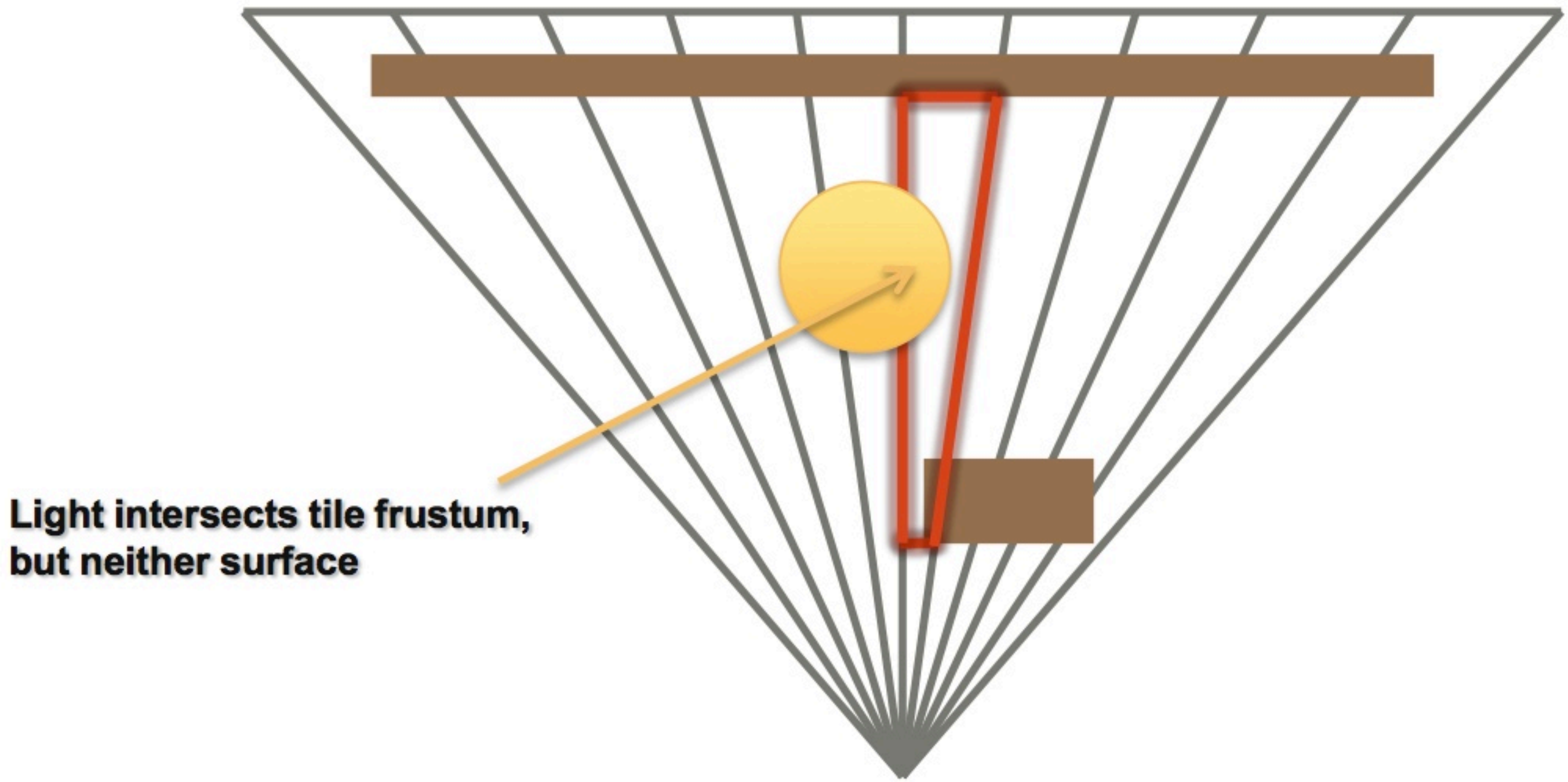


Number of lights evaluated per G-buffer sample

(scene contains 1024 point lights)

Image Credit: A. Lauritzen

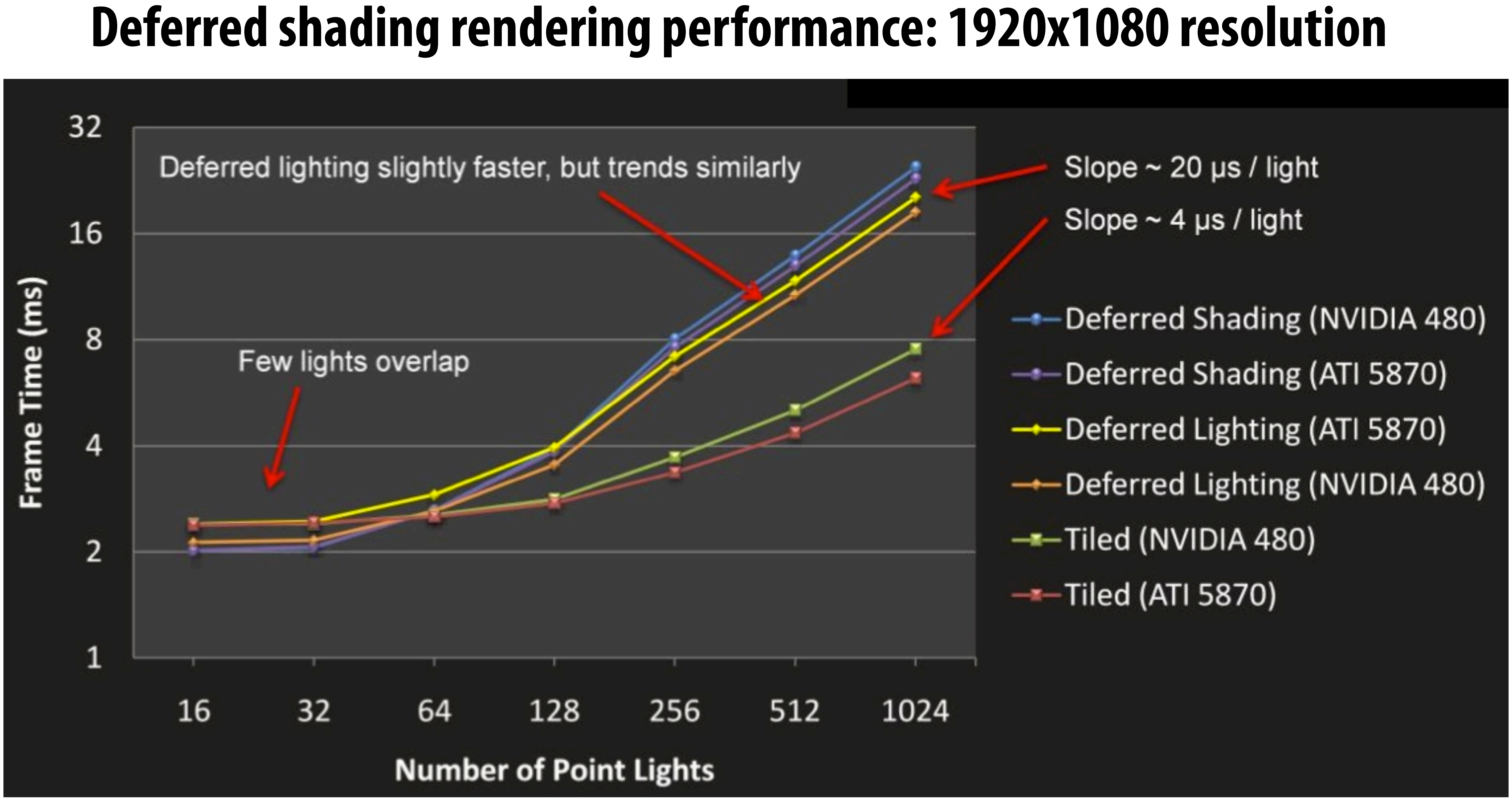
Culling inefficiency near silhouettes



Tile screen boundaries + tile (zmin, zmax) define a frustum
Depth bounds are not tight when tile contains an object silhouette

Image Credit: A. Lauritzen

Tiled vs. conventional deferred shading



[Lauritzen 2009]

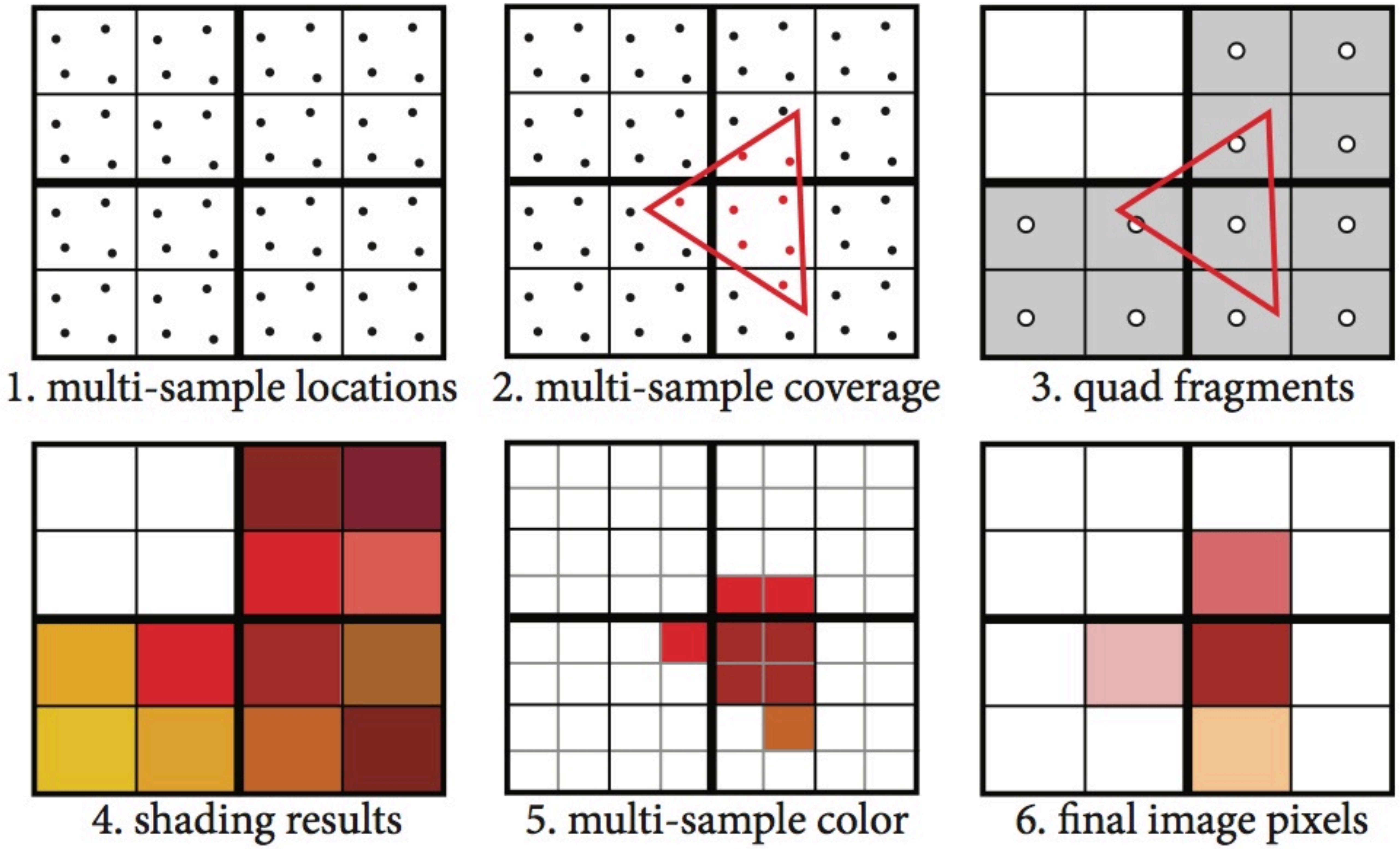
"Forward plus" rendering

- Tile based light culling is not specific to deferred shading
- "Forward+" rendering:

```
Phase 1: Render Z-prepass to populate depth buffer
Phase 2: In compute shader: compute zmin/zmax for all tiles, compute light lists
Phase 3: Render scene with shading enabled:
            Fragment shader determines tile containing fragment
            Shader uses tile's light list when computing surface illumination.
```

- Achieves light culling benefits of tiled-deferred approach in a forward renderer
    - Primary difference is how shading is scheduled:
        - Forward+ recomputes shading inputs using a second geometry pass. ("rematerialization"). Rasterizer generates shading work.
        - Tiled-deferred stores shading inputs in G-buffer. Application iterates over samples using compute shader to generate shading work.

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 shades exactly once per G-buffer sample ***
- **MSAA: shades once per triangle contributing coverage to samples in a pixel**
  - For pixels in interior of projected triangle: one shading computation per pixel
  - Extra shading occurs at pixels along triangle boundaries
    - This is desirable: extra shading necessary to anti-alias object silhouettes
    - Undesirable consequence is extra shading when two adjacent triangles from the same surface surface meet.

*** This is also why transparency is challenging in a deferred shading system**

Two anti-aliasing solutions for deferred shading

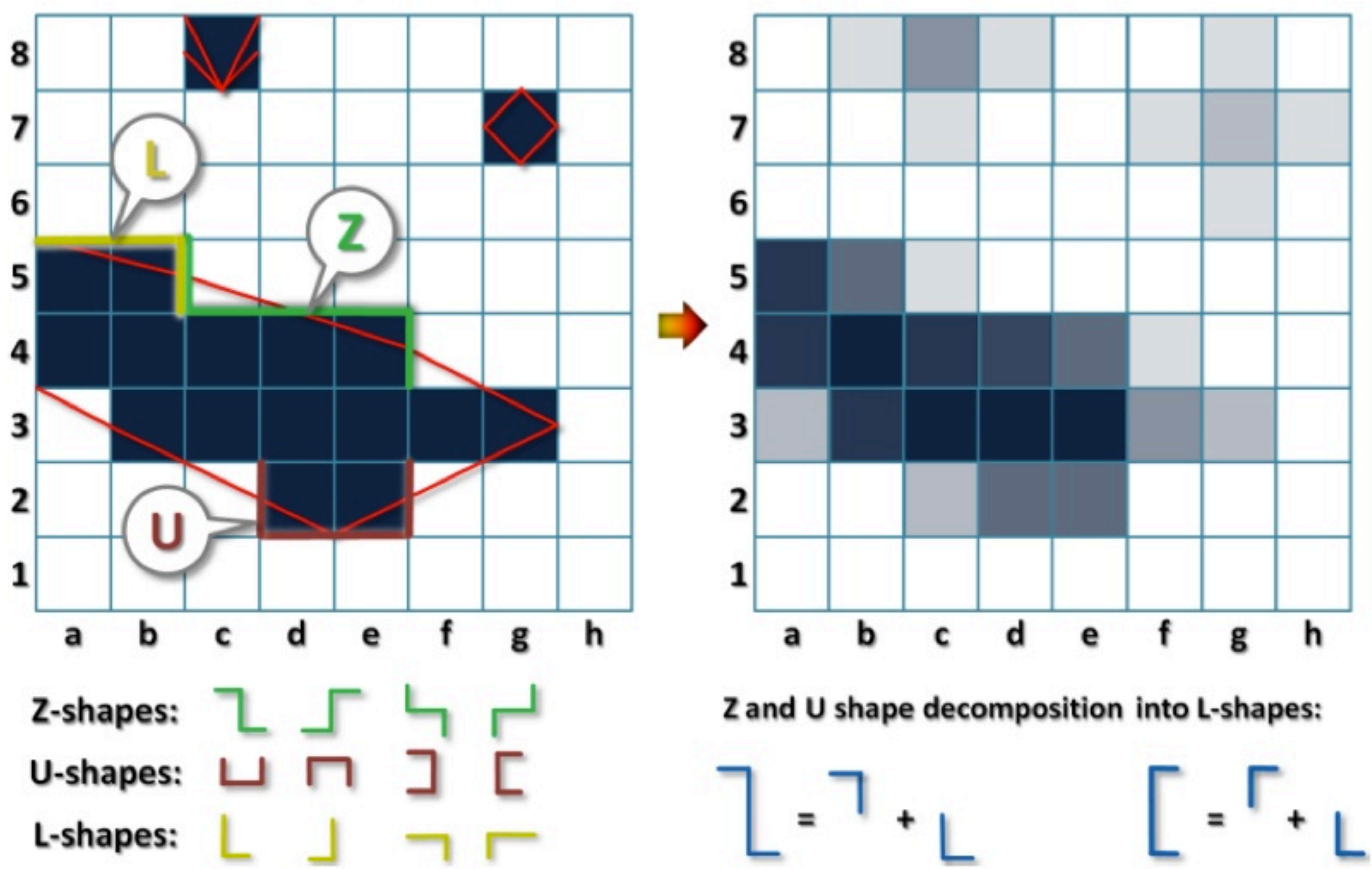
- **Super-sample G-buffer**
  - Generate super-sampled G-buffer
  - Shade at G-buffer resolution
  - Resample shaded results to get final frame-buffer pixels
  - Problems:
    - Increased G-buffer footprint (store "fat pixels" at super-sampled resolution
      - 1900 x 1200 x 4spp x 20 bytes per sample = 173 MB frame-buffer
    - Increased shading cost (shade at visibility rate, not once per pixel!)

- **Intelligently filter aliased shading results**
  - Does not increase G-buffer footprint or shading cost, produces artifacts
  - Current popular technique: morphological anti-aliasing (MLAA)

Morphological anti-aliasing (MLAA)

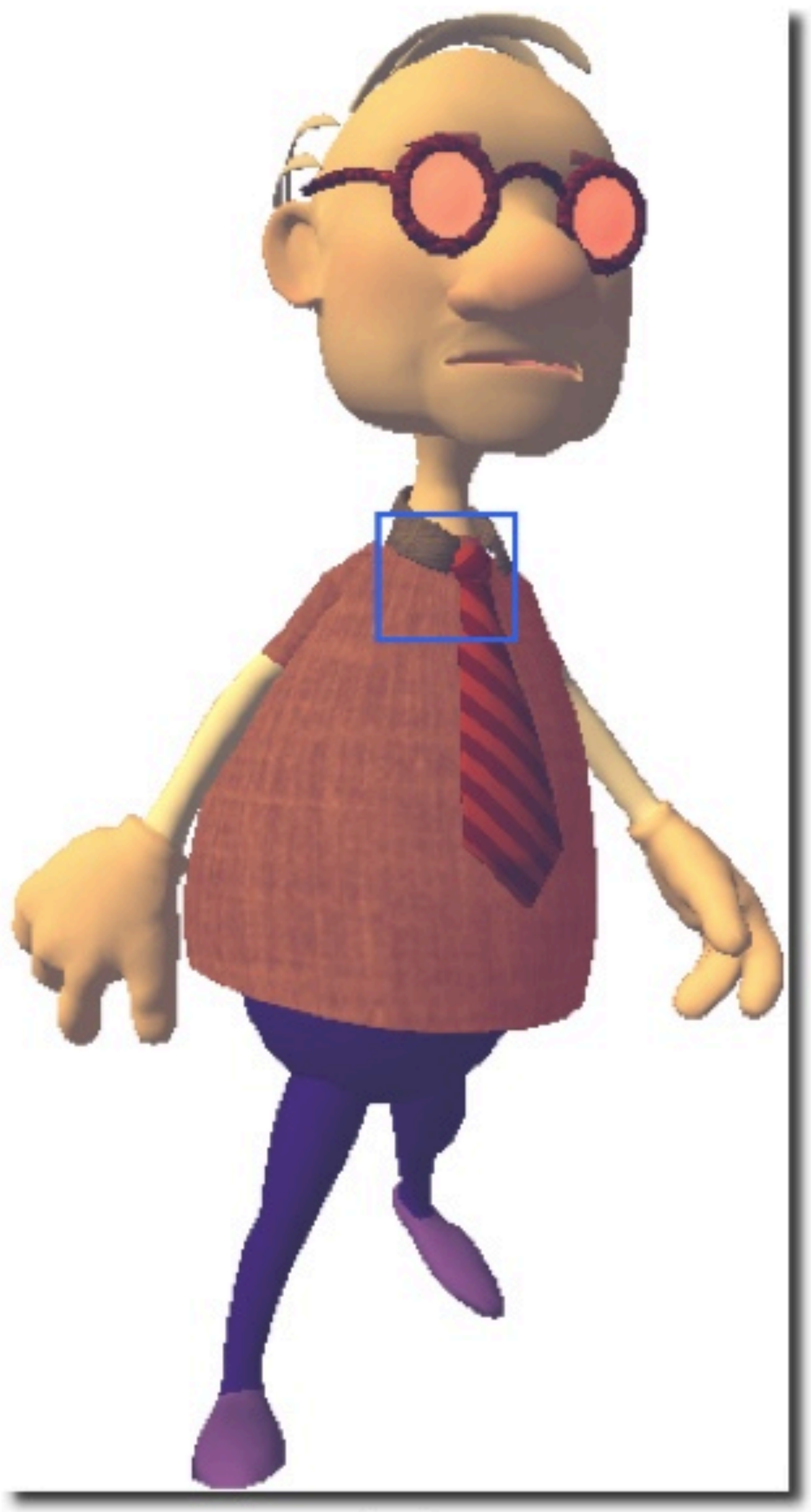
[Reshetov 09]

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



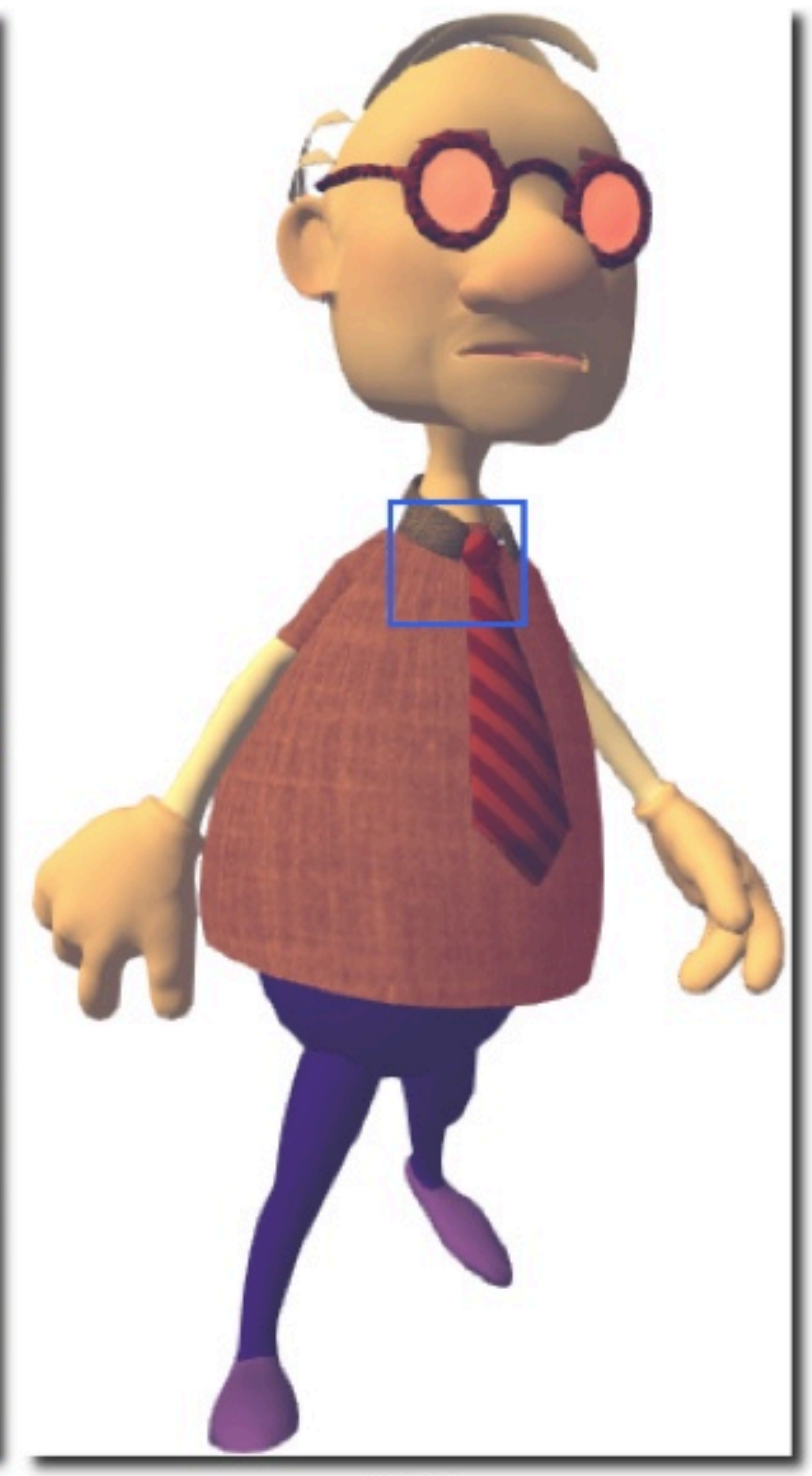
Morphological anti-aliasing (MLAA)

[Reshetov 09]



Aliased image
(one shading sample per pixel)

Zoomed views
(top: aliased, bottom: after MLAA)



After filtering using MLAA

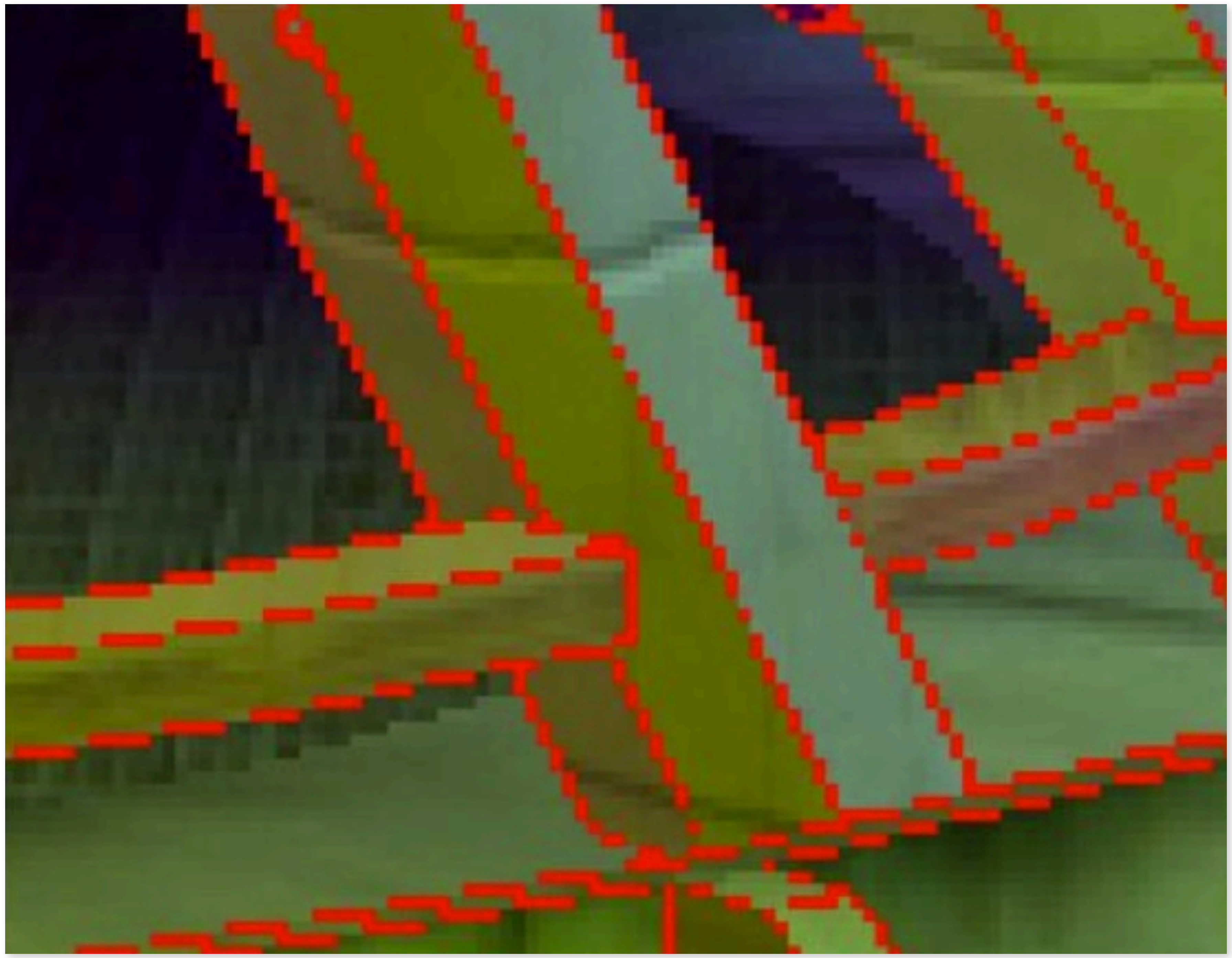
Anti-aliasing solutions for deferred shading

- **Super-sample G-buffer, super-sample shading**
  - Increases G-buffer footprint and shading cost
- **Intelligently filter aliases shading results (MLAA popular choice)**
  - Does not increase G-buffer footprint or shading costs, but may produce artifacts (hallucinates edges/detail)
- **Application implements MSAA on its own**
  - Render super-sampled G-buffer
  - Launch one shader instance for each G-buffer pixel, not each sample
  - Shader implementation:

```
Detect if pixel contains an edge  // how might this be done without geometry information?
If pixel contains edge:
    Shade all G-buffer samples for pixel (sequentially in shader)
    Combine results into single per pixel color output
else:
    Shade one G-buffer sample, store result
```

  - Increases G-buffer footprint, approximately same shading cost as MSAA
  - Some additional BW cost (to detect edges) + potential execution divergence in shader

Handling divergence when implementing MSAA in a shader



Red pixels = shader determines these pixels contain edges (require additional shading)

Adaptive shading rate 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:

e.g., multi-pass solution:

- **Phase 1: categorize pixels, set stencil buffer**
- **Phase 2: shade pixels requiring 1 shading computation**
- **Phase 3: flip stencil value, shade pixels requiring N shading computations**

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

Deferred shading in mobile GPUs

- **Energy-efficient rendering**
  - Philosophy: aggressive cull unnecessary work to conserve energy
- **Implementation of OpenGL ES graphics pipeline by imagination PowerVR GPUs is sort-middle tiled (just like assignment 1) with deferred shading**
  - Note: this is deferred shading implemented by the system, not on top of the graphics pipeline by the application
  - Tiled rendering implementation can circumvent problem of large G-buffer footprint

Phase 2 implementation of tiled renderer: (bin processing)

```
For each bin:
   For each triangle in bin's triangle list:
      Rasterize triangle (also store triangle id per sample in frame buffer)

   // Determine quad-fragments that contribute to frame buffer
   For each sample in tile:
      Given triangle id, compute quad fragment that contributed to sample
      Add quad-fragment to list of quad fragments to shade (if not in list already)

   // Shade only quad-fragments that contribute coverage
   For each required quad-fragment:
      Shade quad-fragment and contribute results into frame-buffer
```

Deferred shading summary

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

- Modern motivations
  - Scaling scenes to complex lighting conditions (many lights, diverse lights)
  - High geometric complexity (due to tessellation) increases overhead of Z-prepass
  - Yet another motivation: tiny triangles increase overhead of quad-fragment-based forward shading
- 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 footprint characteristics
    - Trade light data footprint for G-buffer footprint
  - Different bandwidth characteristics
  - Different execution coherence 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: not used for transparent surfaces

Final comments

- **Which is better, forward or deferred shading?**
  - Depends on context
  - Is geometric complexity high? (prepass might be costly)
  - Are triangles small? (forward shading has overhead)
  - Is multi-sample anti-aliasing desired? (G-buffer footprint might be too large)
  - Is there significant divergence impacting lighting computations?

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

- **One lesson from today: when considering new techniques or a new system design, be cognizant of interoperability with existing features and optimizations**
  - Deferred shading is not compatible with hardware-accelerated MSAA implementations (application must role its own version of MSAA... and still takes a large G-buffer footprint hit)
  - Deferred shading does not support transparent surfaces

Reading

- *A Sort-Based Deferred Shading Architecture for Decoupled Sampling*. P. Clarberg et al. SIGGRAPH 2013