Lecture 17:
Modern Rendering Techniques Using the Graphics Pipeline

Interactive Computer Graphics
Stanford CS248, Winter 2019
Left over from last time...
Data-driven texture synthesis

- Input: low resolution texture image
- Want: high resolution texture that appears “like” the input
Non-parametric texture synthesis

[Efros and Leung 99]

Source textures

Synthesized Textures

Increasing neighborhood search window

5x5 11x11 15x15 23x23
Algorithm: non-parametric texture synthesis

Main idea: given NxN neighborhood \( w(p) \) around unknown pixel \( p \), want probability distribution function for value of \( p \), given \( w(p) \).

For each pixel \( p \) to synthesize:

1. Find other patches in the image that are similar to the NxN neighborhood around \( p \)
2. Center pixel of patches are candidates for \( p \)
3. Randomly sample from candidates weighted by distance \( d \)

[Efros and Leung 99]
More texture synthesis examples

Source textures

Synthesized Textures

Naive tiling solution

[Efros and Leung 99]
Image completion example

Original Image

Masked Region

Completion Result

Image credit: [Barnes et al. 2009]
Problem: low performance

- Large patch windows + full image search = slow
  - Large windows: preserve structure
  - Full-image search: highly relevant examples are rare

- Must repeat search process for all pixels

- Possible accelerations
  - Limit search window
  - Use acceleration structure for search (e.g., k-d tree)
  - Dimensionality reduction of patches + approximate nearest neighbor search
  - Exploit image coherence
PatchMatch

- A randomized algorithm for rapidly finding correspondences between image patches

- Problem definition:
  - Given images A and B, for each overlapping patch in image A, compute the offset to the nearest neighbor patch in image B
  - Overlapping patches: each patch defined by its center pixel (ignoring boundary conditions, each MxN image consists of MxN patches)
  - PatchMatch computes “nearest neighbor field” (NNF)
    - NNF is function \( f: A \rightarrow \mathbb{R}^2 \) (maps patches in A to patches in B)
    - Example: if patch \( b \) in image B is NN of patch \( a \) in image A, then \( f(a) = b \)
PatchMatch idea #1

- Law of large numbers: a non-trivial fraction of a large field of random offset assignments are likely to be good guesses
- Initialize $f$ with random values

Visualization of $f$:

Saturation = magnitude of match offset (gray indicates matching patch in B is at same pixel location as match patch in A)

Hue = direction of offset
offset $X$ = red-cyan axis
offset $Y$ = blue-yellow axis

Image credit: [Barnes et al. 2009]
PatchMatch idea #2: spatial coherence

- High coherence of nearest neighbors in natural images
- Nearest neighbor of patch at \((x,y)\) should be a strong hint for where to find nearest neighbor of patch at \((x+1,y)\)

How this graph was made:
1. Compute NNF for collection of images
2. For select pixels \((x,y)\), compare NN offset to NN offsets of adjacent pixels \((x-1,y), (x+1,y), (x,y-1), (x,y+1)\)

Image credit: [Barnes et al. 2009]
Propagation: improving the NNF estimate

- The NNF estimate provides a “best-so-far” NN for each patch in A
  - NN patch: \( f(a) \)
  - NN distance = \( d(a,b) \) (where \( b = f(a) \))

- Try to improve NNF estimate by exploiting spatial coherence with left and top neighbor:
  - Let \( a = (x,y) \), then candidate matches for \( a \) are:
    - \( f(x-1, y) + (1,0) \)
    - \( f(x, y-1) + (0,1) \)
  - If candidate patch is better match than \( f(a) \), then replace \( f(a) \) with candidate
    - Replace \( f(a) \) with candidate patch if \( d(a, f(x,y-1)+(0,1)) < d(a, f(a)) \)

- Next iteration, use bottom and right neighbors as candidates
**PatchMatch iterative improvement**

Experiment:
Reconstruct A using patches from B

Image A

Image B (source of patches)

Random init: 1/4 through iter 1

End of iter 1

Iter 2

Iter 5

Image credit: [Barnes et al. 2009]

Stanford CS248, Winter 2019
Random search: avoiding local minima

- Propagation can cause PatchMatch to get stuck in local minima.

- Sample random sequence of candidates from exponential distribution:
  - Let $a = (x, y)$, then candidate matches for $a$ are: $(x, y) + w\alpha^i R^i$
  - $R^i$ is uniform random offset in $[-1,1] \times [-1,1]$
  - $w$ is maximum search radius (e.g., width of entire image)
  - $\alpha$ is typically $1/2$
  - Check all candidates where $w\alpha^i \geq 1$
Example applications

Photoshop's Content Aware Fill

Object Manipulation

Building segment marked by user

Building scaled up, preserving texture

Image credits: [Barnes et al. 2009]
Back to today’s lecture...
Recall: OpenGL/Direct3D graphics pipeline

- **Operations on vertices**
  - Vertex stream
- **Operations on primitives**
  - Primitive stream
- **Fragment Generation (Rasterization)**
  - Fragment stream
- **Operations on fragments**
  - Shaded fragment stream
- **Operations on screen samples**
  - Screen sample operations (depth and color)

Input: vertices in 3D space

1. Vertices in positioned in normalized coordinate space
2. Triangles positioned on screen
3. Fragments (one fragment per covered sample)
4. Shaded fragments

Output: image (pixels)
Theme of this part of the lecture:

A surprising number of advanced lighting effects can be efficiently approximated using the basic primitives of rasterization pipeline, without the need to actually ray trace the scene geometry:

- Rasterization
- Texture mapping
- Depth buffer for occlusion
Shadows
Shadows

Image credit: Grand Theft Auto V
How to compute if a surface is in shadow?
Review: How to compute if a surface is in shadow?

- Based on ray tracing...
- Trace ray from point $P$ to location $L_i$ of light source
- If ray hits scene object before reaching light source... then $P$ is in shadow
Shadow mapping version (recall Assignment 3)

[Williams 78]

1. Place camera at position of a point light source
2. Render scene to compute depth to closest object to light along uniformly distributed “shadow rays” (answer stored in depth buffer)
3. Store precomputed shadow ray intersection results in a texture

“Shadow map” = depth map from perspective of a point light. (Stores closest intersection along each shadow ray in a texture)

Image credits: Segal et al. 92, NVIDIA
Result of shadow texture lookup approximates visibility result when shading fragment at $P$

Precomputed shadow rays shown in red: Distance to closest object in scene is precomputed and stored in texture map (“shadow map”)
Shadow aliasing due to shadow map undersampling

Shadows computed using shadow map

Correct hard shadows (result from computing $v(x';x'')$ directly using ray tracing)

Image credit: Johnson et al. TOG 2005
Soft shadows

Hard shadows
(created by point light source)

Soft shadows
(created by ???)
Shadow cast by an area light
Percentage closer filtering (PCF) — hack!

- Instead of sample shadow map once, perform multiple lookups around desired texture coordinate
- Tabulate fraction of lookups that are in shadow, modulate light intensity accordingly

### Hard Shadows
(one lookup per fragment)

### PCF Shadows
(16 lookups per fragment)

<table>
<thead>
<tr>
<th>Shadow Map (consider case where distance from light to surface is 0.5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>0 0 0 0 0 0 1 1 1</td>
</tr>
<tr>
<td>0 0 0 0 0 1 1 1 1</td>
</tr>
<tr>
<td>0 0 0 0 1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 0 1 1 1 1 1 1</td>
</tr>
<tr>
<td>0 0 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>0 1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 1 1 1 1 1 1 1 1</td>
</tr>
</tbody>
</table>
What PCF computes 😥
Ambient occlusion
Ambient occlusion

Idea:
Precompute “amount of hemisphere” that is occluded within distance $d$ from a point. When shading, attenuate environment lighting by this amount.
“Screen-space” ambient occlusion in games

1. Render scene to depth buffer
2. For each pixel $p$ (“ray trace” the depth buffer to estimate occlusion of hemisphere - use a few samples per pixel)
3. Blur the occlusion map to reduce noise
4. Shade pixels, darken direct environment lighting by occlusion amount

[Diagram showing depth buffer values and pixel $p$]
Ambient occlusion

Direct Lighting (no self-shadowing computations)

Lighting modulated by occlusion
Reflections
Reflections
Recall: perfect mirror reflection
Rasterization: “camera” position can be reflection point

Environment mapping: place ray origin at reflective object

Yields approximation to true reflection results. Why?

Cube map: stores results of approximate mirror reflection rays

(Question: how can a glossy surface be rendered using the cube-map)

Center of projection

Scene rendered 6 times, with ray origin at center of reflective box (produces “cube-map”)
Interreflections
Diffuse interreflections

Why is this gray wall tinted red?

Why is this point not black?

Image credit: Henrik Wann Jensen
Precomputed lighting

- Precompute lighting for a scene offline (possible for static lights)
  - Offline computations can perform advanced shadowing, inter reflection computations

- “Bake” results of lighting into texture map
Precomputed lighting in Unity

Image credit: Unity / Alex Lovett

Visualization of light map texture coordinates
Growing interest in real-time ray tracing

- I’ve just shown you an array of different techniques for approximating different advanced lighting phenomenon

- Challenges:
  - Different algorithm for each effect (code complexity)
  - Algorithms may not compose
  - They are approximations to the physically correct solution (“hacks!”)

- Traditionally, tracing rays to solve these problems was too costly for real-time use
  - That may be changing soon…

This image was ray traced in real-time on a (very high end) GPU

Learn more in CS348B!
Deferred Shading
"Forward" rendering

Typical use of fragment processing stage: evaluate application-defined function from surface inputs to surface color (reflectance)
Deferred shading: two steps

Step 1: Do not use traditional pipeline to generate RGB image

Fragment shader now outputs surface properties (future shading 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 (called a “g-buffer”, for geometry buffer)
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 + depth (32-bits per pixel, per buffer)

<table>
<thead>
<tr>
<th>R8</th>
<th>G8</th>
<th>B8</th>
<th>A8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth 24bpp</td>
<td>Stencil</td>
<td>Intensity</td>
<td>DS</td>
</tr>
<tr>
<td>Lighting Accumulation RGB</td>
<td>Intensity</td>
<td>Normal Y (FP16)</td>
<td>RT0</td>
</tr>
<tr>
<td>Normal X (FP16)</td>
<td>Spec-Power</td>
<td>Spec-Intensity</td>
<td>RT1</td>
</tr>
<tr>
<td>Motion Vectors XY</td>
<td>Diffuse Albedo RGB</td>
<td>Sun-Occlusion</td>
<td>RT2</td>
</tr>
</tbody>
</table>

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

Intuitive to consider G-buffer as one big render target with “fat” pixels
In the example above: $32 \times 5 = 160$ bits $= 20$ bytes per pixel

96-160 bits per pixel is common in games
## Compressed G-buffer layout

**G-buffer layout in Bungie’s Destiny (2014)**

<table>
<thead>
<tr>
<th>8</th>
<th>8</th>
<th>8</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albedo Color RGB</td>
<td></td>
<td></td>
<td>Ambient Occlusion</td>
</tr>
<tr>
<td>Normal XYZ <em>(Biased Specular Smoothness)</em></td>
<td></td>
<td></td>
<td>Material ID</td>
</tr>
<tr>
<td>Depth</td>
<td></td>
<td></td>
<td>Stencil</td>
</tr>
</tbody>
</table>

- **Material information is compressed using indirection**
  - Store material ID in G-buffer
  - Material parameters other than albedo *(specular shape/roughness/color)* stored in table indexed by material ID

*Source: N Tatarchuk: SIGGRAPH 2014 Courses, Matt Hoffman*
Two-pass deferred shading algorithm

- **Pass 1: G-buffer generation pass**
  - Render complete scene geometry using traditional pipeline
  - Write visible geometry information to G-buffer

  After all geometry processing is done...

- **Pass 2: shading/lighting pass**

  For each G-buffer sample \((x,y)\):
  - Read G-buffer data for current sample \((x,y)\)
  - Compute shading by accumulating contribution to reflectance of all lights
  - Output final surface color for sample \((x,y)\)

  Shading/lighting computations are “deferred” until all geometry processing is complete...

Image Credit: J. Klint, “Deferred Rendering in Leadworks Engine”
Why is deferred shading so popular in modern games?
Motivation: why deferred shading?

- Two performance reasons:

- Shading is expensive: deferred shading shades only visible fragments
  - Exactly one shade per output screen sample, regardless of the number of triangles in the scene (minimal amount of work + predictable shading performance that is independent of scene size or triangle submission order)

- Forward rendering shades small triangles inefficiently
GPUs shade at the granularity of 2x2 fragments
(“quad fragment” is the minimum granularity of rasterization output and shading)

Enables cheap computation of texture coordinate differentials
(cheap: derivative computation leverages shading work that must be done by adjacent fragment anyway)

All quad fragments are shaded independently
(communication is between fragments in a quad fragment, no communication required between quad fragments)
Implication: multiple fragments get shaded for pixels near triangle boundaries

Shading computations per pixel
Small triangles result in extra shading

Shaded quad fragments per pixel
(early-z is enabled + scene rendered in approximate front-to-back order to minimize extra shading due to overdraw)

100 pixel-area triangles  10 pixel-area triangles  1 pixel-area triangles

Want to sample appearance approximately once per surface per pixel (assuming correct texture filtering)
But graphics pipeline generates at least one appearance sample per triangle per pixel (actually more, considering quad fragments)
Motivation: why deferred shading?

- Shade only visible surface fragments
- Forward rendering shades small triangles inefficiently (quad-fragment granularity)
- Scalability to increasingly complex lighting environments
1000 lights

[J. Andersson, SIGGRAPH 2009 Beyond Programmable shading course talk]
Forward rendering: naive multiple-light shader

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

// uniform values (read-only inputs to all fragments)
uniform sampler2D myTex;
uniform sampler2D myEnvMaps[MAX_NUM_LIGHTS];
uniform sampler2D myShadowMaps[MAX_NUM_LIGHTS];
LightDefinition lightList[MAX_NUM_LIGHTS];
int numLights;

// fragment shader receives surface normal and texture coords uv
varying vec3 norm;
varying vec3 uv;

void main() {
    vec3 kd = texture2d(myTex, uv);
    vec4 result = vec4(0, 0, 0, 0);
    for (int i=0; i<numLights; i++) {
        result += ... // eval contribution of light to surface reflectance here
    }

    gl_FragColor = result; // output color of fragment shader
}
```
Rendering as a triple “for” loop

Naive forward rasterization-based renderer:

initialize $z_{\text{closest}}[]$ to INFINITY  // store closest-surface-so-far for all samples
initialize color[]  // store scene color for all samples
bind all relevant light data in buffers: light descriptors, shadow maps, etc.

for each triangle $t$ in scene:  // loop 1: triangles
    $t_{\text{proj}} = \text{project}_\text{triangle}(t)$
    for each sample $s$ in frame buffer:  // loop 2: visibility samples
        if ($t_{\text{proj}}$ covers $s$)
            for each light $l$ in scene:  // loop 3: lights
                accumulate contribution of light $l$ to surface appearance
                if (depth of $t$ at $s$ is closer than $z_{\text{closest}}[s]$)
                    update $z_{\text{closest}}[s]$ and color[$s$]

Efficient rasterization techniques (tiled, hierarchical, bounding boxes) serve to reduce $T \times S$ complexity of finding covered samples.
Rendering as a triple “for” loop

Naive forward rasterization-based renderer:

initialize z_closest[] to INFINITY  // store closest surface-so-far for all samples
initialize color[]                 // store scene color for all samples
bind all relevant shadow maps, etc.
for each triangle t in scene:      // loop 1: triangles
    t_proj = project_triangle(t)
    for each sample s in frame buffer: // loop 2: visibility samples
        if (t_proj covers s)
            for each light l in scene:    // loop 3: lights
                accumulate contribution of light l to surface appearance
                if (depth of t at s is closer than z_closest[s])
                    update z_closest[s] and color[s]

F x L loop: # fragments x # lights
In practice: not all lights illuminate all surfaces
Would like to be more efficient in computing these interactions
(just like we were efficient computing triangle/visibility sample interactions)
Naive many-light shader with culling

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

// uniform values (read-only inputs to all fragments)
uniform sampler2D myTex;
uniform sampler2D myEnvMaps[MAX_NUM_LIGHTS];
uniform sampler2D myShadowMaps[MAX_NUM_LIGHTS];
LightDefinition lightList[MAX_NUM_LIGHTS];
int numLights;

// fragment shader receives surface normal and texture coords uv
varying vec3 norm;
varying vec3 uv;

void shader() {
  vec3 kd = texture2D(myTex, uv);
  vec4 result = float4(0, 0, 0, 0);
  for (int i=0; i<numLights; i++)
  {
    if (this fragment is illuminated by current light)
    {
      if (lightList[i].type == SPOTLIGHT)
        result += // eval contribution of light here
      else if (lightList[i].type == POINTLIGHT)
        result += // eval contribution of light here
      else if ...
    }
  }
  gl_FragColor = result; // output color
}
```

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

SIMD 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(F x L) work
F = number of fragments
L = number of lights

Stanford CS248, Winter 2019
Forward rendering: techniques for scaling to many lights

- **Goal:** avoid performing $F \times L$ “is-illuminated” checks

- **One solution:** application maintains per-object light lists
  - Each scene object maintains list of lights that illuminate it
  - CPU computes this list each frame by intersecting light volumes with scene geometry
    (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
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 many small batches
  - First generate data structures for all lights: e.g., shadow maps
  - Before drawing each object, only send data for relevant lights to graphics pipeline
  - Write different variants of shader that are specialized for different numbers of lights (4-light version, 8-light version...)

- Implications:
  - Good: very efficient shaders with fewer conditionals
  - Bad: many “small” draw comments to sent to GPUs
Recall: rendering as a triple for-loop

Naive forward rasterization-based renderer:

initialize $z_{\text{closest}}[]$ to $\text{INFINITY}$  // store closest surface-so-far for all samples
initialize color[]                      // store scene color for all samples
bind all relevant shadow maps, etc.

for each triangle $t$ in scene:    // loop 1: triangles
    $t_{\text{proj}} = \text{project\_triangle}(t)$
    for each sample $s$ in frame buffer: // loop 2: visibility samples
        if ($t_{\text{proj}}$ covers $s$)
            for each light $l$ in scene:    // loop 3: lights
                accumulate contribution of light $l$ to surface appearance
                if (depth of $t$ at $s$ is closer than $z_{\text{closest}}[s]$)
                    update $z_{\text{closest}}[s]$ and color[$s$]
Reordering triangles for light coherence

Shader code is now specialized to exactly 4 lights:

```
initialize z_closest[] to INFINITY  // store closest surface-so-far for all samples
initialize color[]                  // store scene color for all samples
bind all relevant shadow maps, etc.

for each group of triangles with the same number of lights:  // loop 0: groups of triangles
  bind specific shader for number of lights
  for each triangle t in group:                           // loop 1: triangles
    t_proj = project_triangle(t)
    for each sample s in frame buffer:                   // loop 2: visibility samples
      if (t_proj covers s)
        for lights 0 through 3:                           // loop 3: lights (specialized for 4 lights)
          accumulate contribution of light l to surface appearance
          if (depth of t at s is closer than z_closest[s])
            update z_closest[s] and color[s]
```
"Multi-pass" rendering for light coherence

initialize z_closest[] to INFINITY // store closest surface-so-far for all samples
initialize color[] // store scene color for all samples
assume z buffer is initialized using a z prepass.

for each light l in scene:                          // loop 1: lights
    bind single light shader specific to current light type
    bind relevant shadow map, etc.
    for each triangle t lit by light:                // loop 2: triangles
        t_proj = project_triangle(t)
        for each sample s in frame buffer:           // loop 3: visibility samples
            if (t_proj covers s)
                accumulate contribution of light l to surface appearance  // specialized to 1 light
                if (depth of t == z_closest[s])
                    update color[s]

Reorder loops: draw scene once per light

Each pass, only draw triangles illuminated by current light (per-light object lists)
Shader accumulates illumination of visible fragments from current light into frame buffer
Forward rendering: techniques for scaling to many lights

- Application maintains light lists

- Option 1: draw scene in many small batches
  - First generate data structures for all lights: e.g., shadow maps
  - Compute per-object light lists, before drawing each object, only bind data for relevant lights
  - Precompile specialized shaders for different sets of bound lights (4-light version, etc…)
  - For each object:
    - Render object with specialized shader for relevant lights
  - Good: can use specialized fragment shader for current type/number of lights
  - Bad: many draw comments to GPU (draw comment = single object, or small group of objects with the same number of lights)

- 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)
  - Good: Minimal footprint for light data
  - Good: can use specialized fragment shader for current type/number of lights
  - Bad: significant overheads: redundant geometry processing, many G-buffer accesses, redundant execution of common shading sub-expressions in fragment shader
Basic many light deferred shading algorithm

initialize \( z_{\text{closest}} \) to \( \text{INFINITY} \) // store closest-surface-so-far for all samples
initialize \( g\text{buffer}[] \) // store surface information for all samples

for each triangle \( t \) in scene: // loop 1: triangles
  \( t_{\text{proj}} = \text{project Triangle}(t) \)
  for each sample \( s \) in frame buffer: // loop 2: visibility samples
    if (\( t_{\text{proj}} \) covers \( s \))
      emit geometry information
      if (depth of \( t \) at \( s \) is closer than \( z_{\text{closest}}[s] \))
        update \( z_{\text{closest}}[s] \) and \( g\text{buffer}[s] \)

initialize \( c\text{olor}[] \) // store color for all samples

for each light in scene: // loop 1: lights
  bind single light shader specific to current light type
  bind relevant shadow map, etc.
  for each sample \( s \) in frame buffer: // loop 2: visibility samples
    load \( g\text{buffer}[s] \)
    accumulate contribution of current light to surface appearance into \( c\text{olor}[s] \)

- **Good**
  - Only process scene geometry once (only in phase 1)
  - Outer loop of phase 2 is over lights:
    - Avoids light data footprint issues (stream over lights)
    - Continues to avoid divergent execution in fragment shader
  - Recall other deferred benefits: only shade visibility samples (and no more)

- **Bad?**
Basic many light deferred shading algorithm

```
initialize z_closest[] to INFINITY // store closest-surface-so-far for all samples
initialize gbuffer[] // store surface information for all samples
for each triangle t in scene: // loop 1: triangles
    t_proj = project_triangle(t)
    for each sample s in frame buffer: // loop 2: visibility samples
        if (t_proj covers s)
            emit geometry information
            if (depth of t at s is closer than z_closest[s])
                update z_closest[s] and gbuffer[s]

initialize color[] // store color for all samples
for each light in scene: // loop 1: lights
    bind single light shader specific to current light type
    bind relevant shadow map, etc.
    for each sample s in frame buffer: // loop 2: visibility samples
        load gbuffer[s]
        accumulate contribution of current light to surface appearance into color[s]
```

**Bad:**

- **High G-buffer footprint:** G-buffer has large footprint (especially when G-buffer is supersampled!)
- **High bandwidth costs** (read G-buffer each pass, output to frame buffer)
- Exactly one shading computation per frame-buffer sample
  - Does not support transparency (need multiple fragments per pixel)
  - Supersampling for anti-aliasing becomes expensive
Reducing deferred shading bandwidth costs

- **Batching:** process multiple lights in each phase 2 accumulation pass
  - Amortizes 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 (only generate fragments for covered G-buffer samples)
    - Light-fragment interaction predicate is evaluated by rasterizer, not in shader
  - 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, rendering only the front-facing triangles of the light volume will generate fragments in the yellow shaded region (these are the only g-buffer samples that can be effected by the light)
Scene with 256 lights
Visualization of light-sample interaction count

Per-light culling is performed using a 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)
Screen tiled-based light culling

Main idea: build list of lights that effect each screen tile (not each object)
Project light volume, then intersect in 2D with tiles

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

Image credit: HMREngine: http://www.hmrengine.com/blog/?p=399
Tile-based deferred shading: better light culling efficiency
(16x16 granularity of light culling is apparent in figure)

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

Image Credit: A. Lauritzen
Challenge: anti-aliasing geometry in a deferred renderer
Supersampling in a deferred shading system

- In assignment 1, you anti-aliased rendering via supersampling
  - Stored $N$ color samples and $N$ depth samples per pixel

- Deferred shading makes supersampling challenging due to large amount of information that must be stored per pixel
  - $2800 \times 1800$ (my Mac laptop I’m presenting on today)
  - 4 samples per pixel
  - 20 bytes per G-buffer sample
  
  $= 403$ MB G-buffer

  (24 GB/sec of memory bandwidth just to read and write the G-buffer at 30 fps)
Morphological anti-aliasing (MLAA)

Detect careful designed patterns in rendered image
For detected patterns, blend neighboring pixels according to a few simple rules ("hallucinate" a smooth edge.. it's a hack!)

Note: modern interest in replacing MLAA patterns with DNN-based anti-aliasing.
Morphological anti-aliasing (MLAA)

Aliased image (one shading sample per pixel)
Zoomed views (top: aliased, bottom: after MLAA)
After filtering using MLAA

[Reshetov 09]
Summary: deferred shading

- Very popular technique in modern games
- Creative use of graphics pipeline
  - Create a G-buffer, not a final image
- Two major motivations
  - Convenience and simplicity of separating geometry processing logic/costs from shading costs
  - Potential for high performance under complex lighting and shading conditions
    - Shade only once per sample despite triangle overlap
    - Often more amenable to “screen-space shading techniques”
      - e.g., screen space ambient occlusion