Lecture 21:
Deferred Shading

Visual Computing Systems
CMU 15-769, Fall 2016
Deferred shading

- Popular algorithm used by many modern real-time renderers

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

- Not a new idea: present in several classic graphics systems, but not directly implemented by modern GPU-accelerated graphics pipeline
  - However, modern graphics pipeline provides mechanisms to allow application to implement deferred shading efficiently
  - Natively implemented by PowerVR mobile GPUs
  - Classic hardware-supported implementations:
    - [Deering et al. 88]
    - UNC PixelFlow [Molnar et al. 92]
The graphics pipeline

Vertex Generation → Vertex Processing → Primitive Generation → Primitive Processing → Rasterization (Fragment Generation) → Early Z → Fragment Processing → Frame-Buffer Ops → Frame Buffer

“Feed-forward” rendering

Typical use of fragment processing stage: evaluate application-defined function from surface inputs to surface color (reflectance)
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 (called a “g-buffer”, for geometry buffer)
G-buffer = “geometry” buffer

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)

<table>
<thead>
<tr>
<th>R8</th>
<th>G8</th>
<th>B8</th>
<th>A8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth 24bpp</td>
<td>Lighting Accumulation RGB</td>
<td>Normal X (FP16)</td>
<td>Normal Y (FP16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Spec-Power</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Spec-Intensity</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Sun-Occlusion</td>
</tr>
</tbody>
</table>

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 = 160 bits = 20 bytes per pixel

96-160 bits per pixel is common in games

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

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

- Material information 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

<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>Normal XYZ * (Biased Specular Smoothness)</td>
<td>Ambient Occlusion</td>
<td>Material ID</td>
</tr>
<tr>
<td>Depth</td>
<td></td>
<td></td>
<td>Stencil</td>
</tr>
</tbody>
</table>

Source: N Tatarchuk: SIGGRAPH 2014 Courses
"Two pass" deferred shading

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 (called a “g-buffer”, for geometry buffer)

After all geometry has been rendered, execute shader for each sample in the G-buffer: shader reads geometry information for sample, computes RGB output

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

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

- **Pass 2: shading/lighting 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

Image Credit: J. Klint, “Deferred Rendering in Leadworks Engine”
Motivation: why deferred shading?

- **Shading is expensive:** shade only visible fragments
  - Deferred shading amounts to perfect early occlusion culling
  - Deferred shading is triangle order invariant (will only shade visible fragments, regardless of application’s triangle submission order)
  - Also has nice property that the number of shaded fragments is independent of scene complexity (predictable shading performance)

- **Recall:** forward rendering shades small triangles inefficiently
  - Recall shading granularity is quad fragments: 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

Graphics applications employ many 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: naive multiple-light shader

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

// uniform values (read-only inputs to all shader instances)
sampler mySamp;
Texture2D<float3> myTex;
Texture2D<float> myEnvMaps[MAX_NUM_LIGHTS];
Texture2D<float> myShadowMaps[MAX_NUM_LIGHTS];
LightDefinition lightList[MAX_NUM_LIGHTS];
int numLights;

// fragment shader receives surface normal and texture coords uv
float4 fragment_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++)
    {
        result += ... // eval contribution of light to surface reflectance here
    }
    return 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[\( s \)]

Triangles are outermost loop:
Triangle setup performed once, amortized across many samples
High coherence in shading computations
(fragments are from the same triangle: same shader program, similar data access)

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 \( z_{\text{closest}}[] \) to \( \infty \) \hspace{1cm} // store closest surface-so-far for all samples
initialize color[] \hspace{1cm} // store scene color for all samples
bind all relevant shadow maps, etc.

for each triangle \( t \) in scene: \hspace{1cm} // loop 1: triangles
    \( t_{\text{proj}} = \text{project\_triangle}(t) \)
    for each sample \( s \) in frame buffer: \hspace{1cm} // loop 2: visibility samples
        if (\( t_{\text{proj}} \) covers \( s \))
            for each light \( l \) in scene: \hspace{1cm} // 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[\( s \)]

\( F \times L \) loop: \# fragments \( \times \) \# 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;
    ...
}

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)
        {
            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 ...
        }
    }
    return result;
}
```

**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:
\(O(F \times L)\) work
- \(F\) = number of fragments
- \(L\) = number of lights
(spatial coherence in predicate should exist)
Forward rendering: techniques for scaling to many lights

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

- **Solution:** application maintains per-object light lists
  - Each object stores 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 bind data for relevant lights to pipeline
  - To avoid SIMD divergence: precompile shader variants that are specialized for different sets of bound lights (4-light version, 8-light version...)
  - Low execution divergence during fragment shading
  - Many graphics state changes, small draw command batch sizes (draw command = single object, or group of objects with the same number of lights)
Recall: 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]
Reordering triangles for light coherence

Shader code is specialized to a specific number of lights:

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

for each group of triangles with the same number of lights: \quad // loop 0: groups of triangles
  bind specific shader for number of lights
  for each triangle \( t \) in group: \quad // loop 1: triangles
    \( t_{\text{proj}} = \text{project}_\text{triangle}(t) \)
  for each sample \( s \) in frame buffer: \quad // loop 2: visibility samples
    if (\( t_{\text{proj}} \) covers \( s \))
      for lights 0 through 3: \quad // loop 3: lights (specialized for 4 lights)
        accumulate contribution of light 1 to surface appearance
        if (depth of \( t \) at \( s \) is closer than \( z_{\text{closest}}[s] \))
          update \( z_{\text{closest}}[s] \) 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**
  - Computed conservatively per frame

- **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: low execution divergence during fragment shading
  - Bad: many graphics state changes (draw call = single object, or 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: Low execution divergence during fragment shading
  - 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_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]

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 \texttt{z\_closest[]} to INFINITY  \hspace{1cm} \text{// store closest-surface-so-far for all samples}
initialize \texttt{gbuffer[]}  \hspace{1cm} \text{// store surface information for all samples}

for each triangle \(t\) in scene:\hspace{1cm} \text{// loop 1: triangles}
\hspace{1cm} \texttt{t\_proj = project\_triangle}(t)

for each sample \(s\) in frame buffer:\hspace{1cm} \text{// loop 2: visibility samples}
\hspace{1cm} if (\texttt{t\_proj covers } s)
\hspace{1.5cm} emit geometry information
\hspace{1.5cm} if (\text{depth of } t \text{ at } s \text{ is closer than } \texttt{z\_closest}[s])
\hspace{2cm} update \texttt{z\_closest}[s] and \texttt{gbuffer}[s]

initialize \texttt{color[]}  \hspace{1cm} \text{// store color for all samples}

for each light in scene:\hspace{1cm} \text{// loop 1: lights}
\hspace{1cm} bind single light shader specific to current light type
\hspace{1cm} bind relevant shadow map, etc.

for each sample \(s\) in frame buffer:\hspace{1cm} \text{// loop 2: visibility samples}
\hspace{1cm} load \texttt{gbuffer}[s]
\hspace{1.5cm} accumulate contribution of current light to surface appearance into \texttt{color}[s]

\textbf{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)
  - Challenging to implement MSAA efficiently (more on this to come)
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)

Image Credit: A. Lauritzen
Tile-based deferred shading

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 “shared memory” available in CUDA/computer 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

[Andersson 09]
Tile-based deferred shading

Step 1: Perform G-buffer generation pass to create G-buffer and Z-buffer
Step 2: Shade G-buffer using compute mode GPU execution

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

LightDescription perTileLightList[MAX_LIGHTS]; // stored in group shared memory

All threads cooperatively compute Z-min, Zmax for current tile from z-buffer 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) (Read G-buffer once)
  load G-buffer data for sample
  for each light in perTileLightList: // no divergence across samples
    result += evaluate contribution of light

  store result to appropriate position in frame buffer (Write final pixel color to frame buffer once)
Tiled-based light culling

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)
Culling inefficiency near silhouettes

Light intersects tile frustum, but neither surface

Tile screen boundaries + tile \((z_{\text{min}}, z_{\text{max}})\) define a frustum
Depth bounds are not tight when tile contains an object silhouette

Image Credit: A. Lauritzen
Tiled vs. conventional deferred shading

Deferred shading rendering performance: 1920x1080 resolution

[Lauritzen 2009]
“Forward plus” rendering

- Tile-based (hierarchical) light culling is not unique to deferred shading
- “Forward+” rendering involves three phases

  Phase 1: Render “Z-prepass” to populate depth buffer (process all geometry)
  Phase 2: In compute shader: compute zmin/zmax for all tiles, compute light lists for screen tiles
  Phase 3: Render scene with shading enabled (process all geometry again)
    Fragment shader determines which tile it resides in
    Shader uses tile’s precomputed light list when computing surface illumination

- Achieves light culling benefits of tiled-deferred approach in a forward renderer (it’s just another reordering of the loops!)
  - Primary difference is how shading is scheduled:
    - Forward+ recomputes shading inputs using a second geometry pass (“rematerialization” of shading inputs via extra computation) but stores light lists in memory between phase 2 and phase 3.
    - Tiled-deferred stores shading inputs in the G-buffer. It never stores light lists in off-chip memory (only compute shader shared memory) because the light list is consumed immediately after its construction in the shader.
Challenge: anti-aliasing geometry in a deferred renderer
Recall: multi-sample anti-aliasing (MSAA)

Main idea: decouple shading sampling rate from visibility sampling rate

Depth buffer: stores depth per sample
Color buffer: stores color per sample
Resample color buffer to get final image pixel values
MSAA in a deferred shading system

- Deferred shading performs exactly one shading computation per G-buffer sample *

- MSAA: shades once per triangle contributing coverage to samples in a pixel
  - So the effective shading rate is adaptive
  - For pixels in interior of projected triangle: this is one shading computation per pixel
  - For pixels on boundary of triangles, extra shading occurs
    - This is desirable: extra shading necessary to anti-alias object silhouettes
    - The undesirable consequence of MSAA is extra shading when two adjacent triangles from the same 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 (e.g., 4 samples per pixel)
  - Shade all G-buffer samples
  - Resample shaded results to get final frame-buffer pixels
  - Problems:
    - Increased G-buffer footprint and G-buffer read/write bandwidth (remember: “fat samples” are stored per G-buffer sample)
    - 1900 x 1200 x 4 spp x 20 bytes per sample = 173 MB frame-buffer
    - Increases shading cost because deferred shading systems I described earlier shade at visibility rate, not once per pixel!

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

Detect carefully designed patterns in rendered image.
For detected patterns, blend neighboring pixels according to a few simple rules ("hallucinate" a smooth edge).

[Reshetov 09]
Morphological anti-aliasing (MLAA)

Aliased image
(one shading sample per pixel)

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

After filtering using MLAA

[Reshetov 09]
Anti-aliasing solutions for deferred shading

- **Super-sample G-buffer, super-sample shading**
  - Increases G-buffer footprint and shading cost

- **Intelligently filter aliased 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 (without HW support)**
  - Render super-sampled G-buffer
  - Launch one shader instance for each output image sample, not each G-buffer sample
  - New shader implementation:
    - Detect if pixel contains an edge  // how might this be done without geometry information?
    - If pixel contains an edge:
      - Shade all G-buffer samples for pixel (sequentially in shader)
      - Resample results into single per pixel color output (e.g., using box filter)
    - Else:
      - Shade only one G-buffer sample for this pixel, store result
  - Increases G-buffer footprint, but approximately same shading cost as MSAA
  - Additional BW cost (to detect edges) + potential execution divergence in shader
Handling divergence when implementing MSAA in a shader

Red pixels = 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:

- Phase 1: categorize pixels, set stencil buffer
- Phase 2: shade pixels requiring one 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)
Tile-based deferred shading (TBDR) in modern mobile GPUs

- **Motivation:** energy-efficient rendering
  - Philosophy: aggressive cull unnecessary fragment work to conserve energy

- **Implementation of OpenGL ES graphics pipeline by imagination PowerVR GPUs** is sort-middle tiled with deferred shading
  - Note: deferred shading is implemented as an optimization by the OpenGL system, not on top of the graphics pipeline by the application as discussed so far in this lecture

**Phase 2 implementation of tiled renderer: (bin processing)**

For each bin:
  - For each triangle in bin’s triangle list:
    - Rasterize triangle (store only triangle id per G-buffer sample)

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

  // Shade only fragments that contribute coverage
  - For each fragment that must be shared:
    - Shade fragment and contribute results into frame buffer

G-buffer stored what triangle covers each sample, not the full set of surface properties (these can be computed as needed based on the triangle ID)
Deferred shading summary

- Main idea: perform shading calculations for opaque surfaces 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, so it’s useful to store and reload results of geometry processing (rather than repeat it)
  - 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 samples, for all drawing primitives
  - Different footprint characteristics
    - Trade footprint of scene light data structures 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 a technique 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: memory bandwidth vs. execution coherence
- Another example of relying on high bandwidth to achieve high ALU utilization
- In real-time graphics: typically manifests 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