06 Deferred shading and multi-pass

CS5625 Spring 2018

Steve Marschner

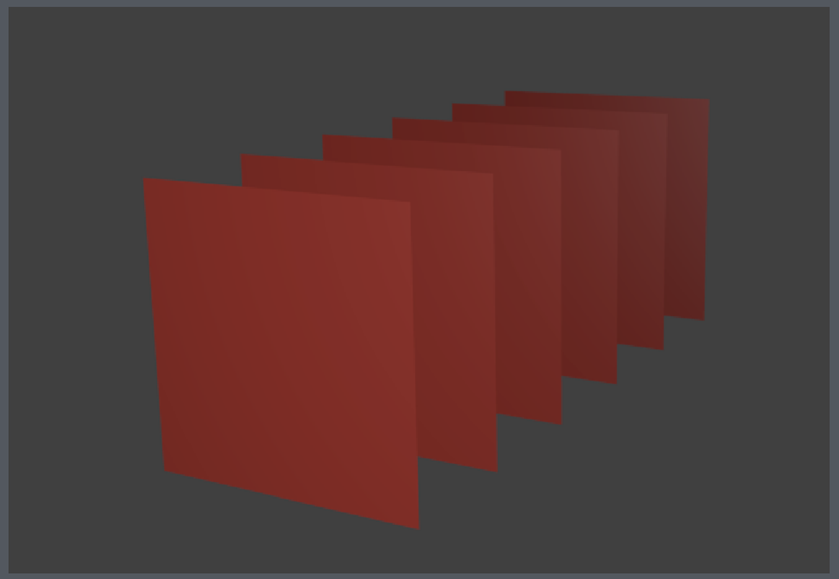
slides adapted from

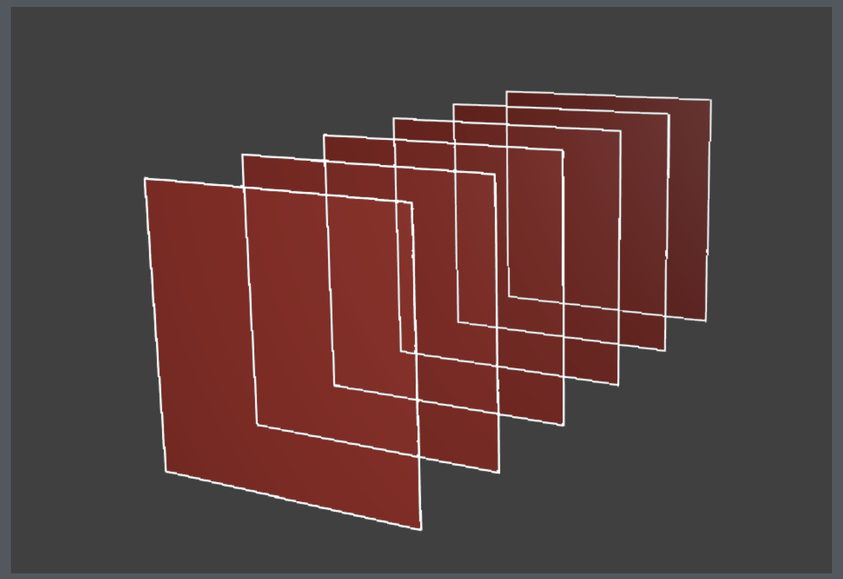
Kavita Bala and Asher Dunn

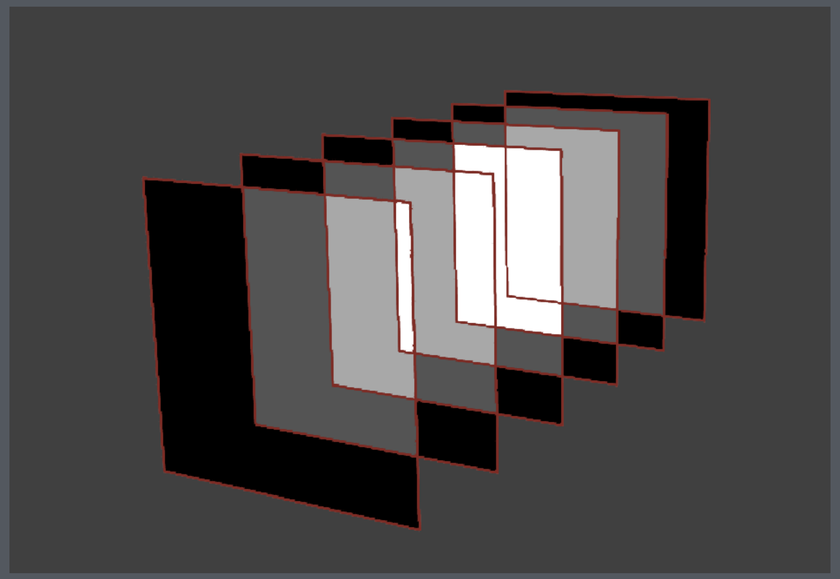
Rendering: forward shading

```
for each object in the scene
   for each triangle in this object
      for each fragment f in this triangle

         gl_FragColor = shade(f)

         if (depth of f < depthbuffer[x, y])
            framebuffer[x, y] = gl_FragColor
            depthbuffer[x, y] = depth of f
         end if

      end for
   end for
end for
```

Problem: Overdraw







Problem: lighting complexity

Just Cause 3 | Avalanche Studios

Missed opportunity: spatial processing

Fragments cannot talk to each other

- a fundamental constraint for performance

Many interesting effects depend on neighborhood and geometry

- bloom
- ambient occlusion
- motion blur
- depth of field
- edge-related rendering effects

Deferred shading approach

First render pass

- draw all geometry
- compute material- and geometry-related inputs to lighting model
- don’t compute lighting
- write shading inputs to intermediate buffer

Second render pass

- don’t draw geometry
- use stored inputs to compute shading
- write to output

Post-processing pass (optional, can also be used with fwd. rendering)

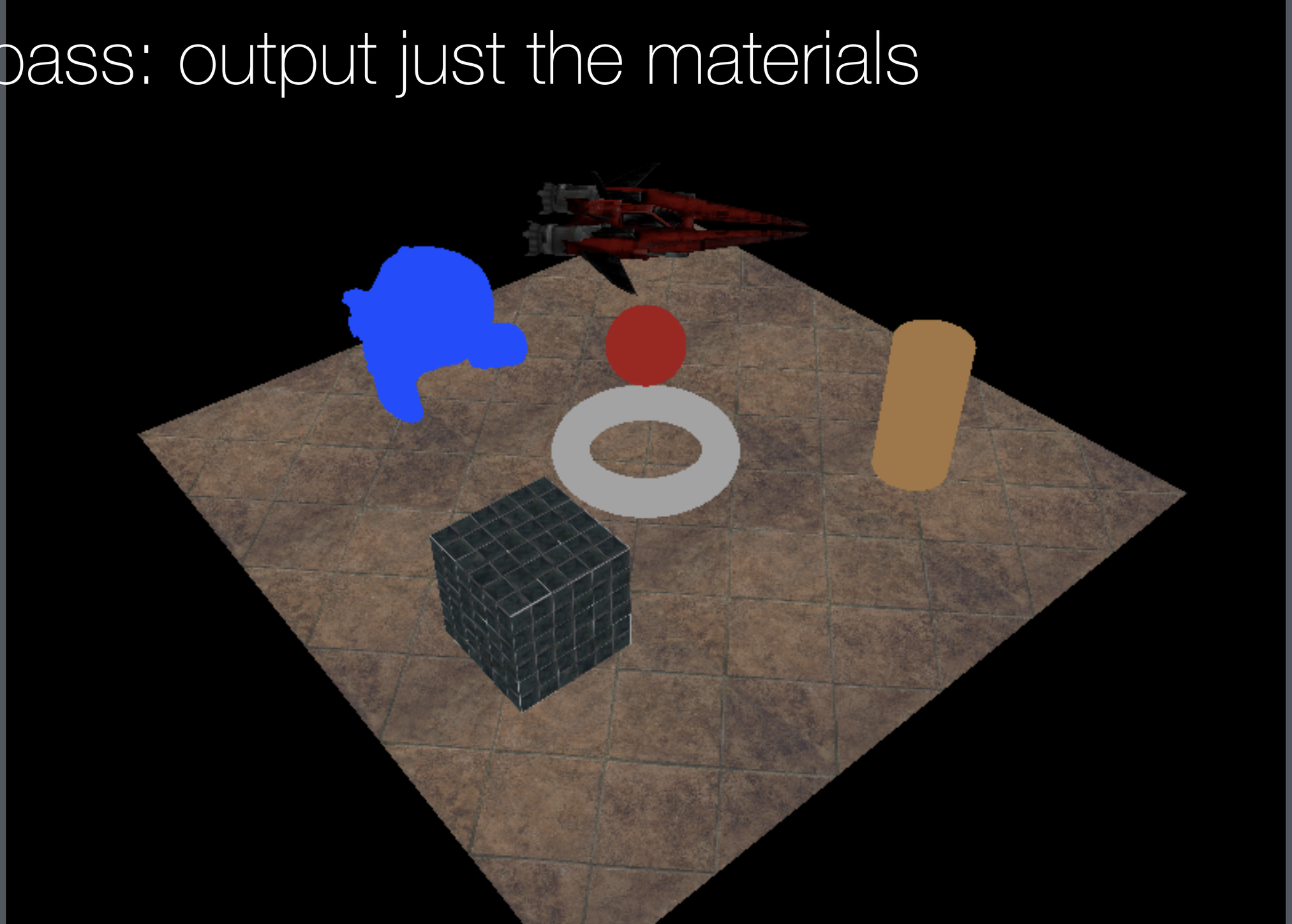
- process final image to produce output pixels

Deferred shading step 1

```
for each object in the scene
   for each triangle in this object
      for each fragment f in this triangle

         gl_FragData[…] = material properties of f
         if (depth of f < depthbuffer[x, y])
            gbuffer[…][x, y] = gl_FragData[…]
            depthbuffer[x, y] = depth of f
         end if

      end for
   end for
end for
```

Here we're making use of an OpenGL feature called "Multiple Render Targets" in which the familiar gl_FragColor is replaced by an array of values, each of which is written to a different buffer.

First pass: output just the materials

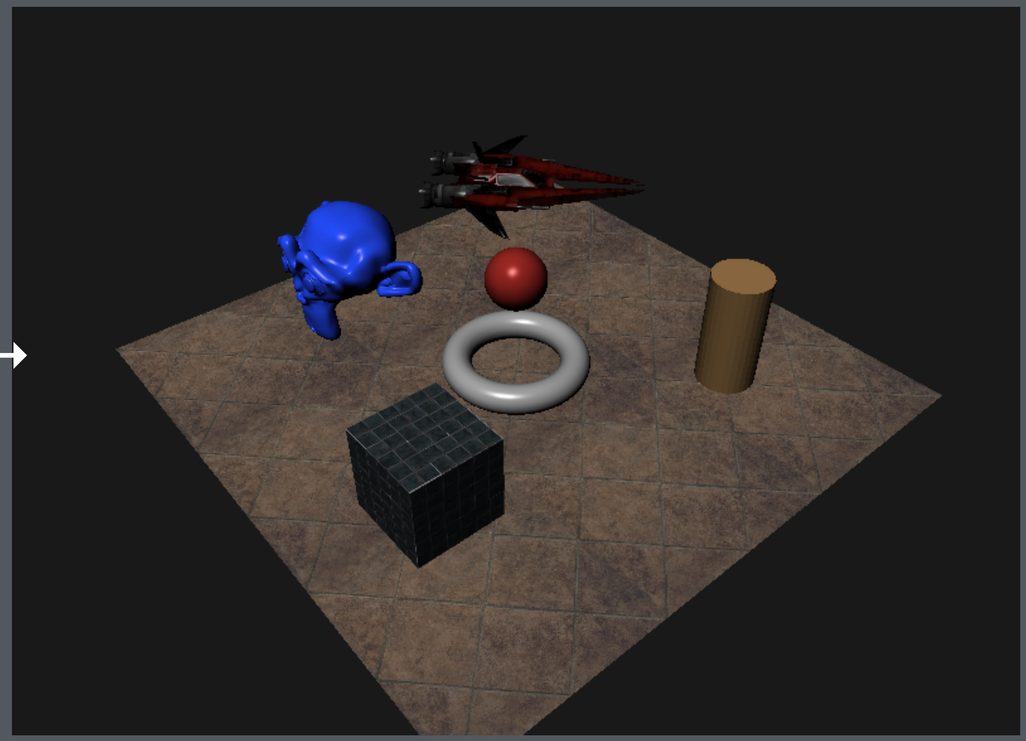


Deferred Shading Step 2

```
for each fragment f in the gbuffer
   framebuffer[x, y] = shade (f)
end for
```

Key improvement: shade (f) only executed for visible fragments.

Output is the same →

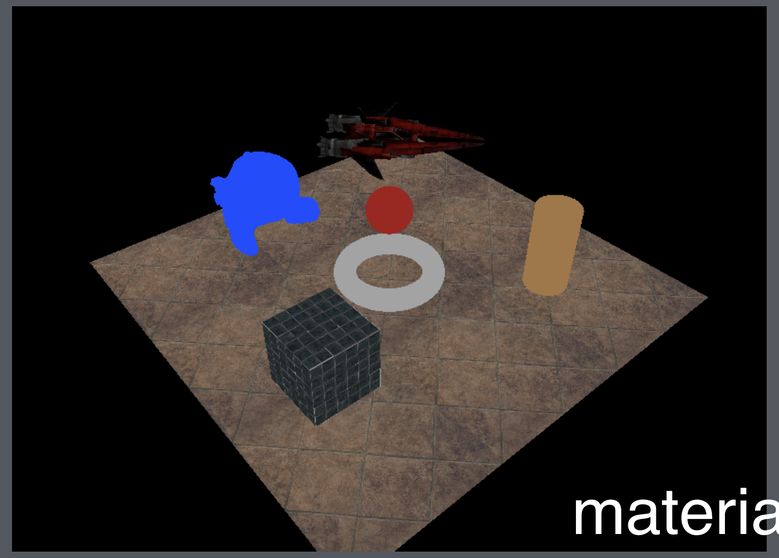


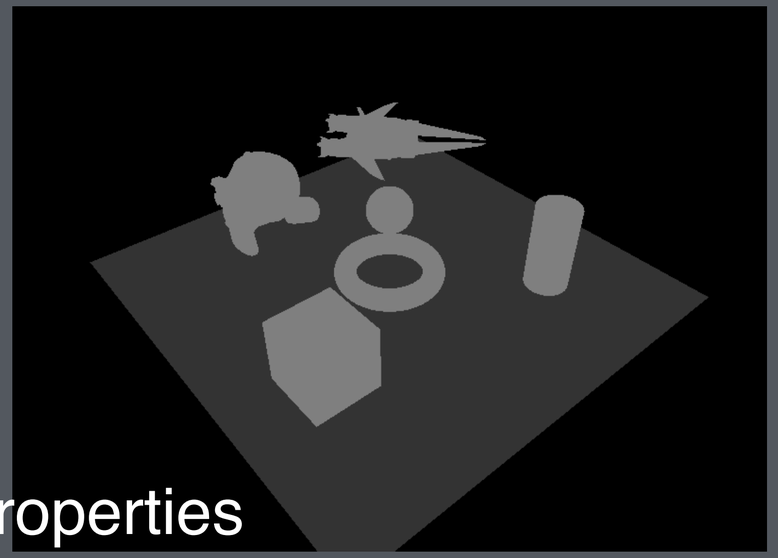
```
for each object in the scene
   for each triangle in this object
      for each fragment f in this triangle

         gl_FragData[…] = material properties of f
         if (depth of f < depthbuffer[x, y])
            gbuffer[…][x, y] = gl_FragData[…]
            depthbuffer[x, y] = depth of f
         end if

      end for
   end for
end for
```

```
for each fragment f in the gbuffer
   framebuffer[x, y] = shade (f)
end for
```

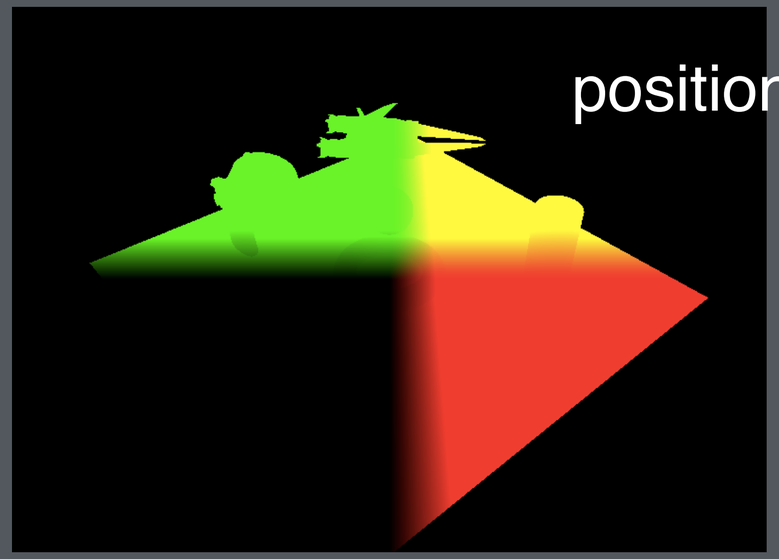
Here we're making use of an OpenGL feature called "Multiple Render Targets" in which the familiar gl_FragColor is replaced by an array of values, each of which is written to a different buffer.

G-buffer: multiple textures

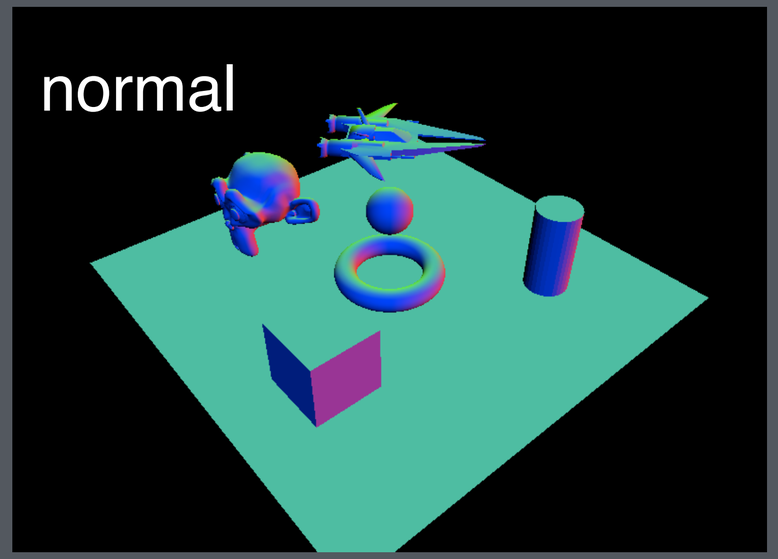




material properties



position



normal

The übershader

Shader which computes lighting based on g-buffer: has code for all material/ lighting models in a single huge shader.

```
shade (f) {
   result = 0;
   if (f is Lambertian) {
      for each light
         result += (n . l) * diffuse;
      end for
   } else if (f is Blinn-Phong) {
      ...
   } else if (f is ...) {
      ...
   }
   return result;
}
```

Übershader inputs

Need access to all parameters of the material for the current fragment:

- Blinn-Phong: kd, ks, n
- Microfacet: kd, ks, alpha
- etc.

Also need fragment position and surface normal

Solution: write all that out from the material shaders:

```
{outputs} = {f.material, f.position, f.normal}
if (depth of f < depthbuffer[x, y])
   gbuffer[x, y] = {outputs}
   depthbuffer[x, y] = depth of f
end if
```

Deferred lighting

Single-pass render has to consider all lights for every fragment

- much wasted effort since only a few lights probably contribute
- batching geometry by which lights affect it is awkward
- straight 2-pass deferred has same problem

With deferred shading, fragments can be visited in subsets

- for each light, draw bounds of (significantly) affected volume
- only compute shading for fragments covered by that
- with depth/stencil games, can only shade fragments inside the volume

Power of Deferred Shading

Can do any image processing between step 1 and step 2!

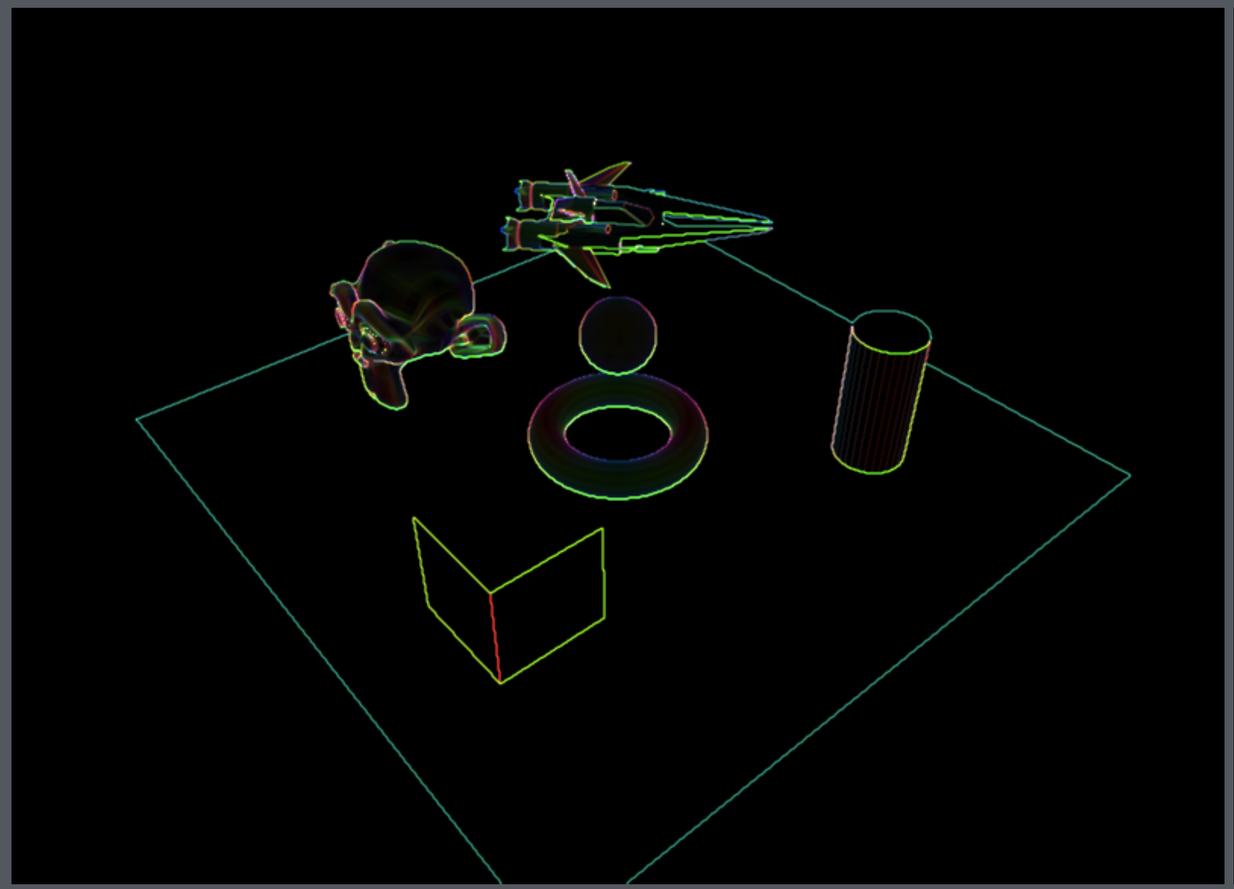
- Recall: step 1 = fill g-buffer, step 2 = light/shade
- Could add a step 1.5 to filter the g-buffer

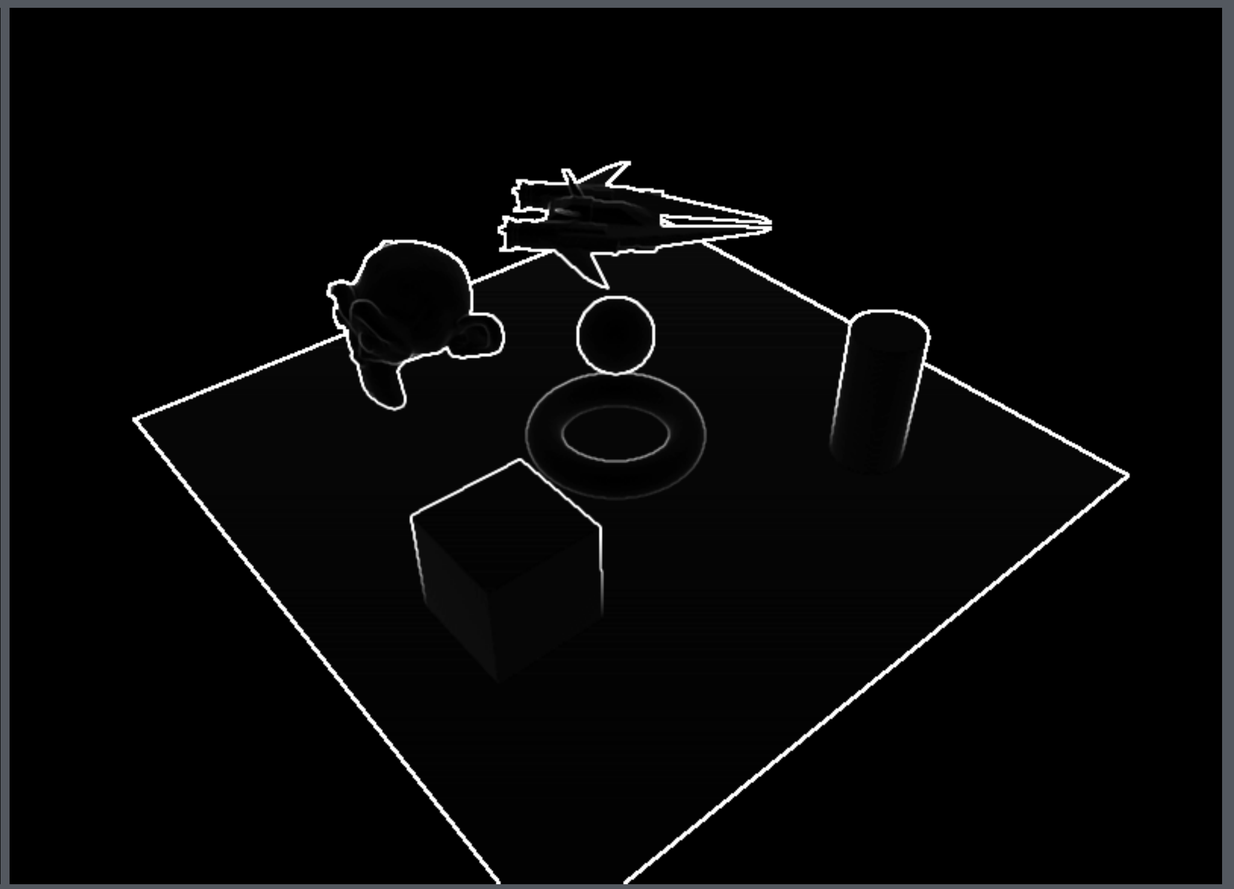
Examples:

- silhouette detection for artistic rendering
- screen-space ambient occlusion
- denoising based on bilateral filter using geometric info

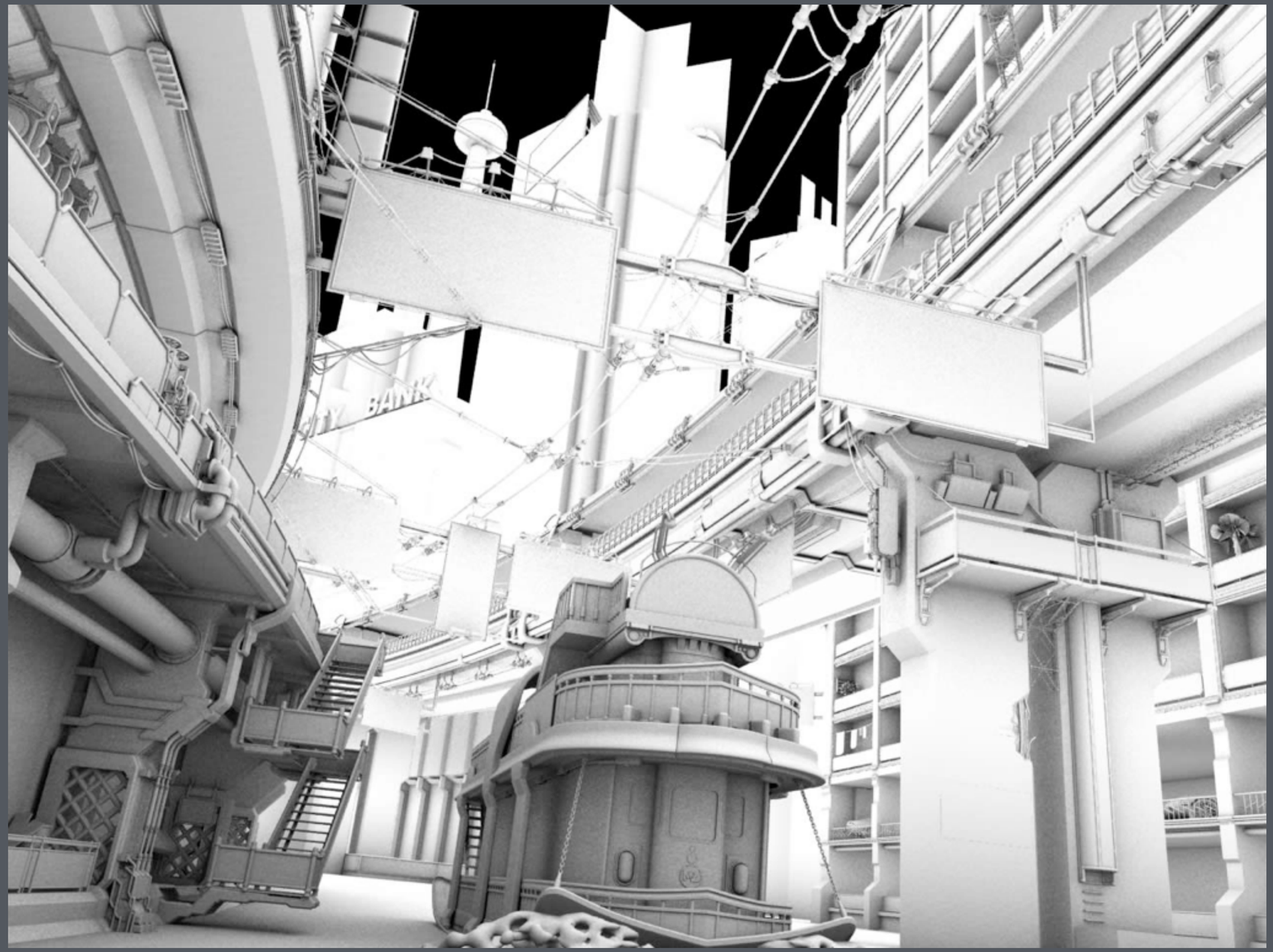
Silhouette detection

By differentiating the depth buffer, can locate silhouettes and creases





Amient occlusion



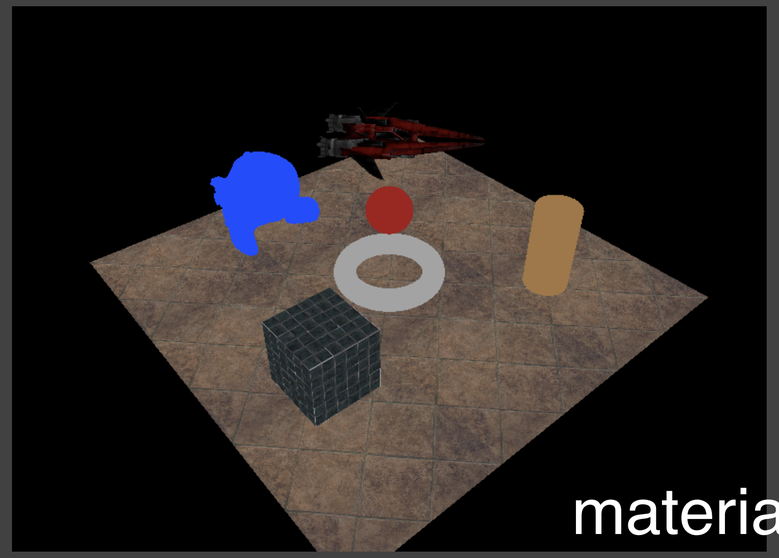
Denoising

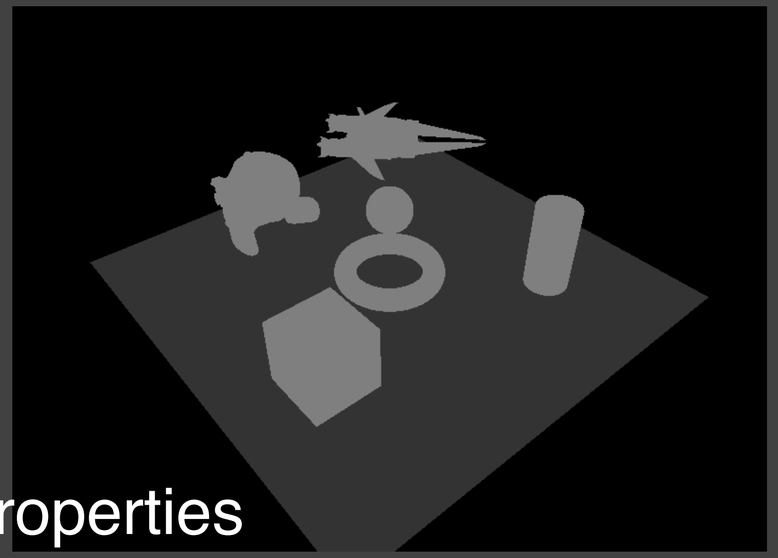
[Mara et al. HPG 2017]

Denoising

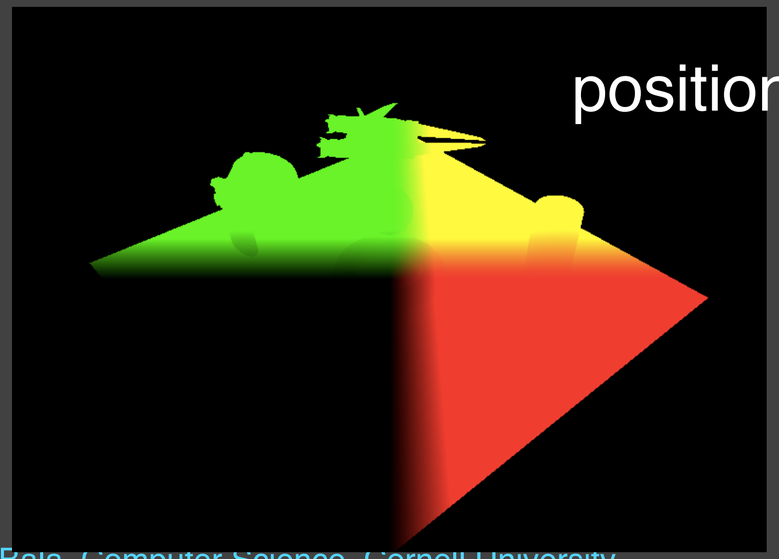
[Mara et al. HPG 2017]

How to Fill the Buffers?

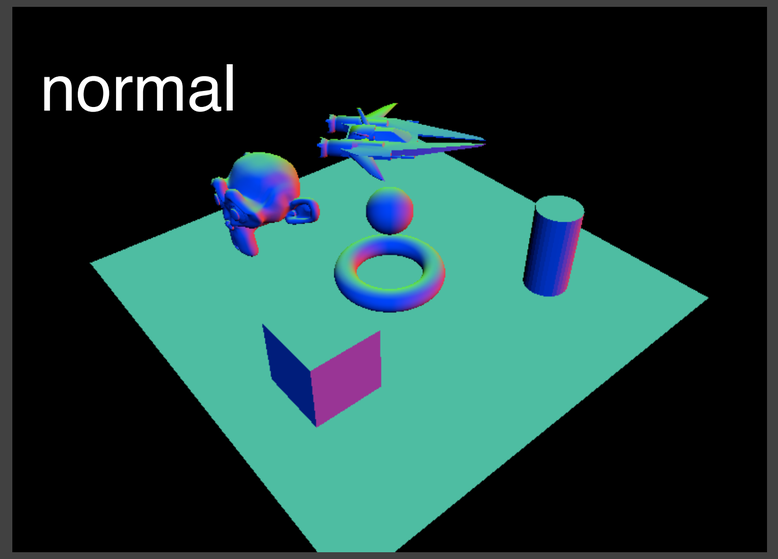




material properties



position



normal

Limitations of Deferred Shading

Each pixel in the g-buffer can only store material and surface info for a single surface.

- blending/transparency is difficult
- antialiasing is a different ball game

For transparency: a “hybrid” renderer

- deferred shading for opaque objects, forward shading for translucent objects
- allows translucent geometry to know about opaque geometry behind it

For antialiasing: smart blurring

- use what is in the g-buffers to blur along but not across edges

Hybrid Rendering

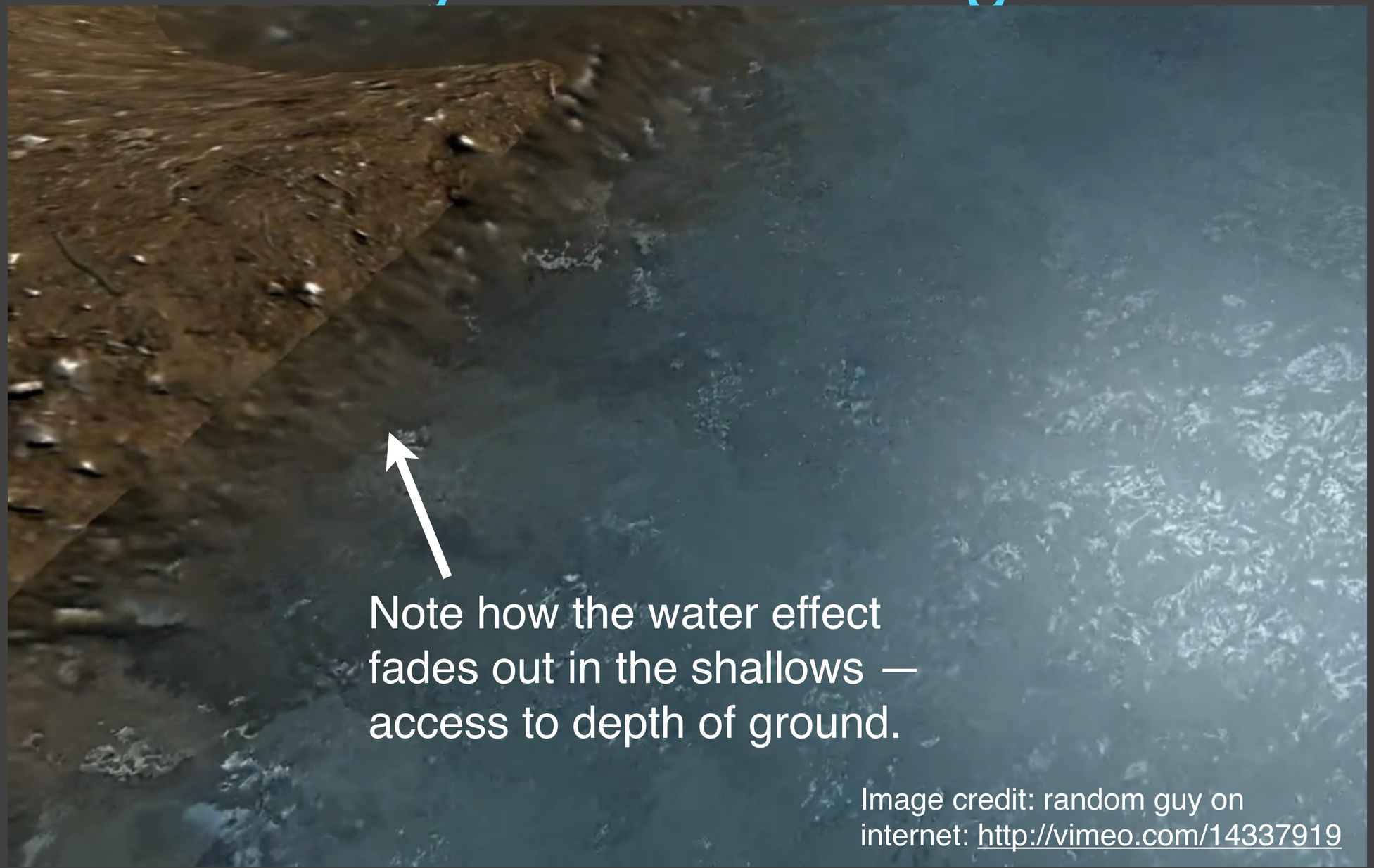


Image credit: random guy on internet: http://vimeo.com/14337919

Antialiasing

Single shading sample per pixel

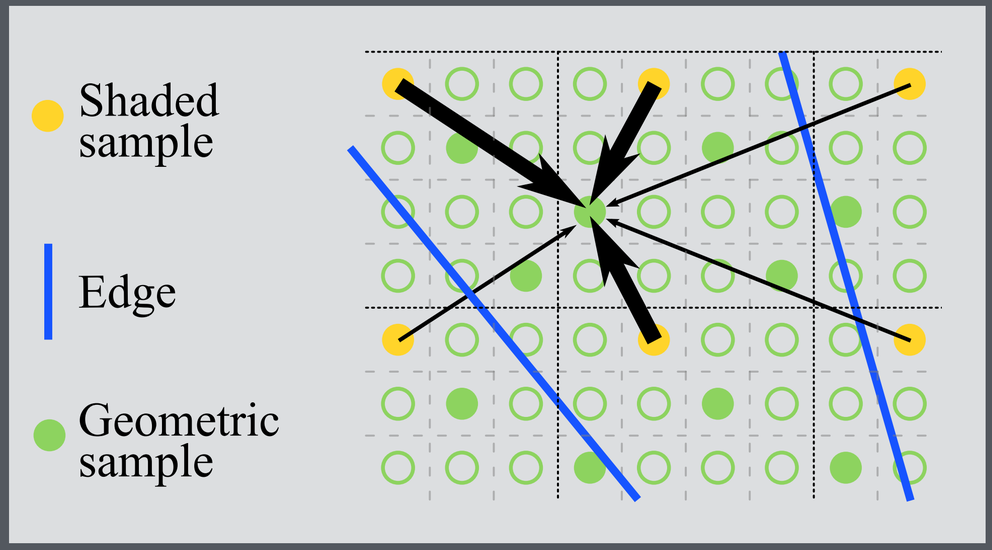
Reconstruct by blending nearby samples

- select them by looking for edges
  (Morphological AA [Reshetov 09])
- learn about edges using multisample depth
  (Subpixel Reconstruction AA [Chajdas et al. 11])

MLAA

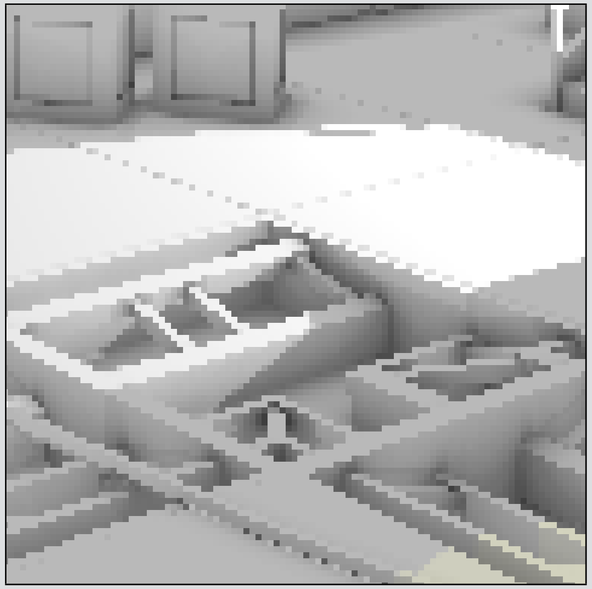
[Reshetov 09]

SRAA

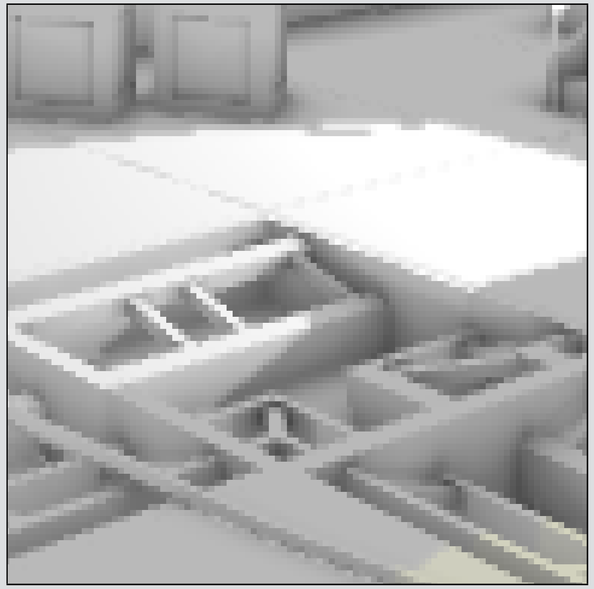


← *Similar Time* →

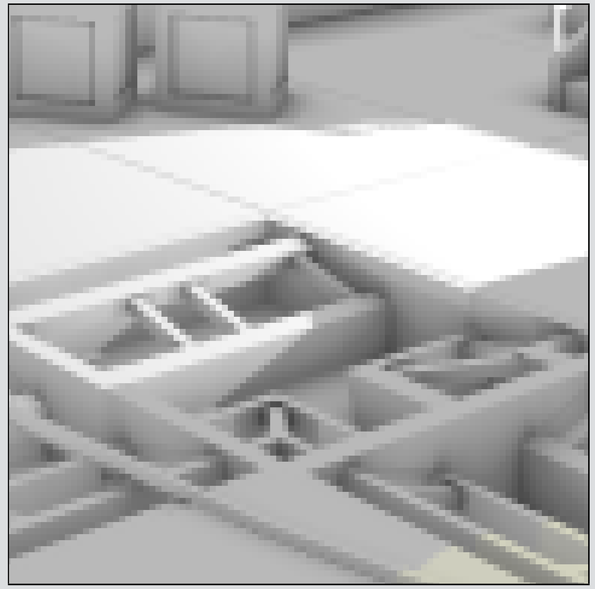
← *Similar Quality* →



(a) 1× Shading + Box (poor, fast)



(b) **NEW: 1× Shading + SRAA (good, fast)**



(c) 16× Shading + Box (good, slow)

[Chajdas et al. 11]

Summary: Deferred Shading

Pros

- Store everything you need in 1st pass
  - normals, diffuse, specular, positions,...
  - G-buffer
- After z-buffer, can shade only what is visible

Cons:

- transparency (only get one fragment per pixel)
- antialiasing (multisample AA not easy to adapt)

Standard game engines provide both forward and deferred paths

How to do all this in OpenGL

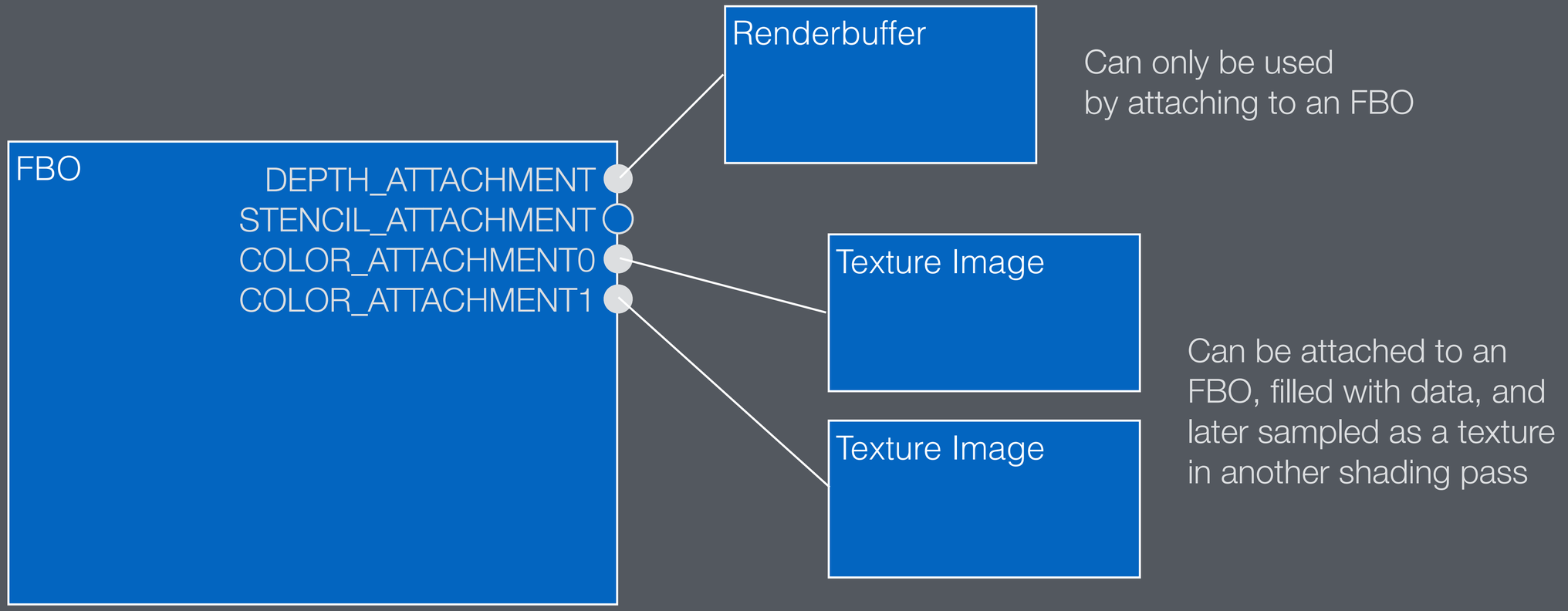
When you first fire up OpenGL, all fragment shader output is written to the framebuffer that shows up in your window

- this is the *default framebuffer*
- it actually can contain multiple buffers: front and back for double-buffering; left and right for stereo/HMD devices. You can control where fragment shader output goes using glDrawBuffer()

For deferred shading and other multi-pass methods, you instead create a (non-default) *Framebuffer Object* (FBO)

- you *attach* images to the FBO to receive fragment shader output
- color attachments (variable number) receive color data from gl_FragData[…] (gl_FragColor is just an alias for gl_FragData[0])
- a depth attachment is required for *z*-buffering to function; stencil attachments are also possible

Framebuffer Object



Can only be used
by attaching to an FBO

Can be attached to an
FBO, filled with data, and
later sampled as a texture
in another shading pass

All these images have to match in size!