09 Deferred shading and postprocessing

CS5625 Spring 2016

Steve Marschner

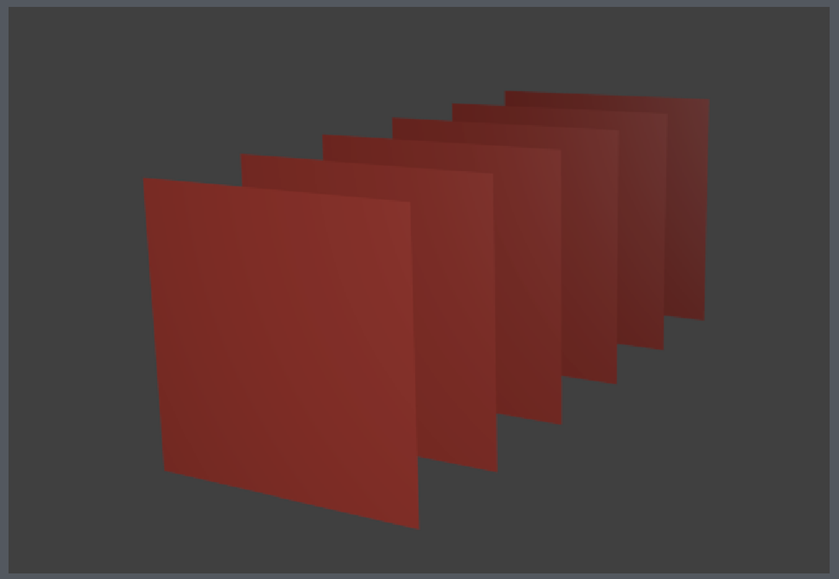
slides adapted from

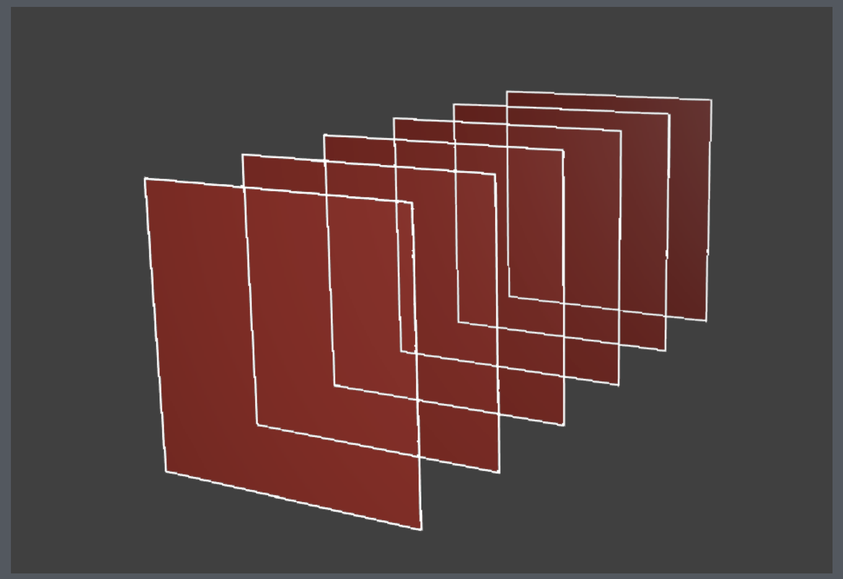
Kavita Bala and Asher Dunn

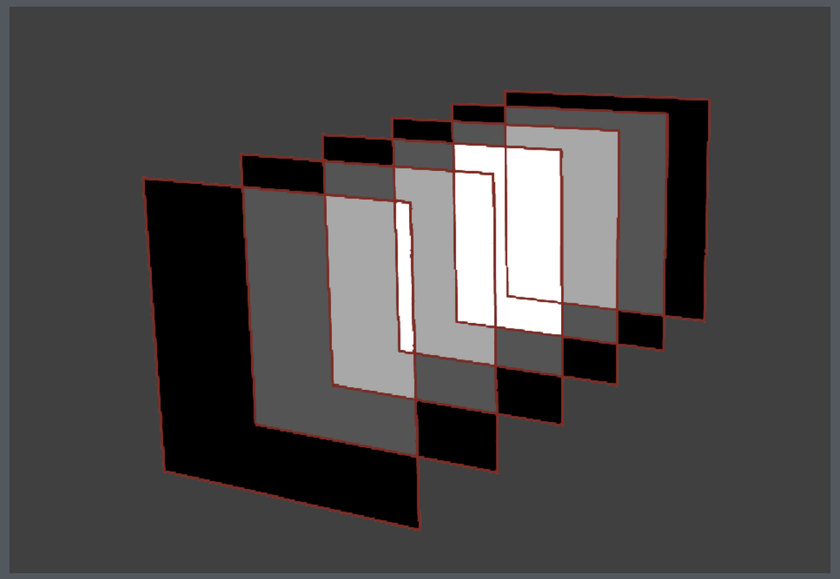
Rendering: forward shading

```
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







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

Buffers galore (from Eric Haines)

A-buffer - Carpenter, 1984
G-buffer - Saito & Takahashi, 1991
M-buffer - Schneider & Rossignac, 1995
P-buffer - Yuan & Sun, 1997
T-buffer - Hsiung, Thibadeau & Wu, 1990
W-buffer - 3dfx, 1996?
Z-buffer - Catmull, 1973 (?)
ZZ-buffer - Salesin & Stolfi, 1989

Accumulation Buffer - Haeberli & Akeley, 1990
Area Sampling Buffer - Sung, 1992
Back Buffer - Baum, Cohen, Wallace & Greenberg, 1986
Close Objects Buffer - Telea & van Overveld, 1997
Color Buffer
Compositing Buffer - Lau & Wiseman, 1994
Cross Scan Buffer - Tanaka & Takahashi, 1994
Delta Z Buffer - Yamamoto, 1991
Depth Buffer - 1984
Depth-Interval Buffer - Rossignac & Wu, 1989
Double Buffer - 1993
Escape Buffer - Hepting & Hart, 1995
Frame Buffer - Kajiya, Sutherland & Cheadle, 1975
Hierarchical Z-Buffer - Greene, 1993
Item Buffer - Weghorst, Hooper & Greenberg, 1984
Light Buffer - Haines & Greenberg, 1986
Mesh Buffer - Deering, 1995
Normal Buffer - Curington, 1985
Picture Buffer - Ollis & Borgwardt, 1988
Pixel Buffer - Peachey, 1987
Ray Distribution Buffer - Shinya, 1994
Ray-Z-Buffer - Lamparter, Muller & Winckler, 1990
Refreshing Buffer - Basil, 1977
Sample Buffer - Ke & Change, 1993
Shadow Buffer - GIMP, 1999
Sheet Buffer - Mueller & Crawfis, 1998
Stencil Buffer - 1992
Super Buffer - Gharachorloo & Pottle, 1985
Super-Plane Buffer - Zhou & Peng, 1992
Triple Buffer
Video Buffer - Scherson & Punte, 1987
Volume Buffer - Sramek & Kaufman, 1999

Deferred shading approach

First render pass

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

Second render pass

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

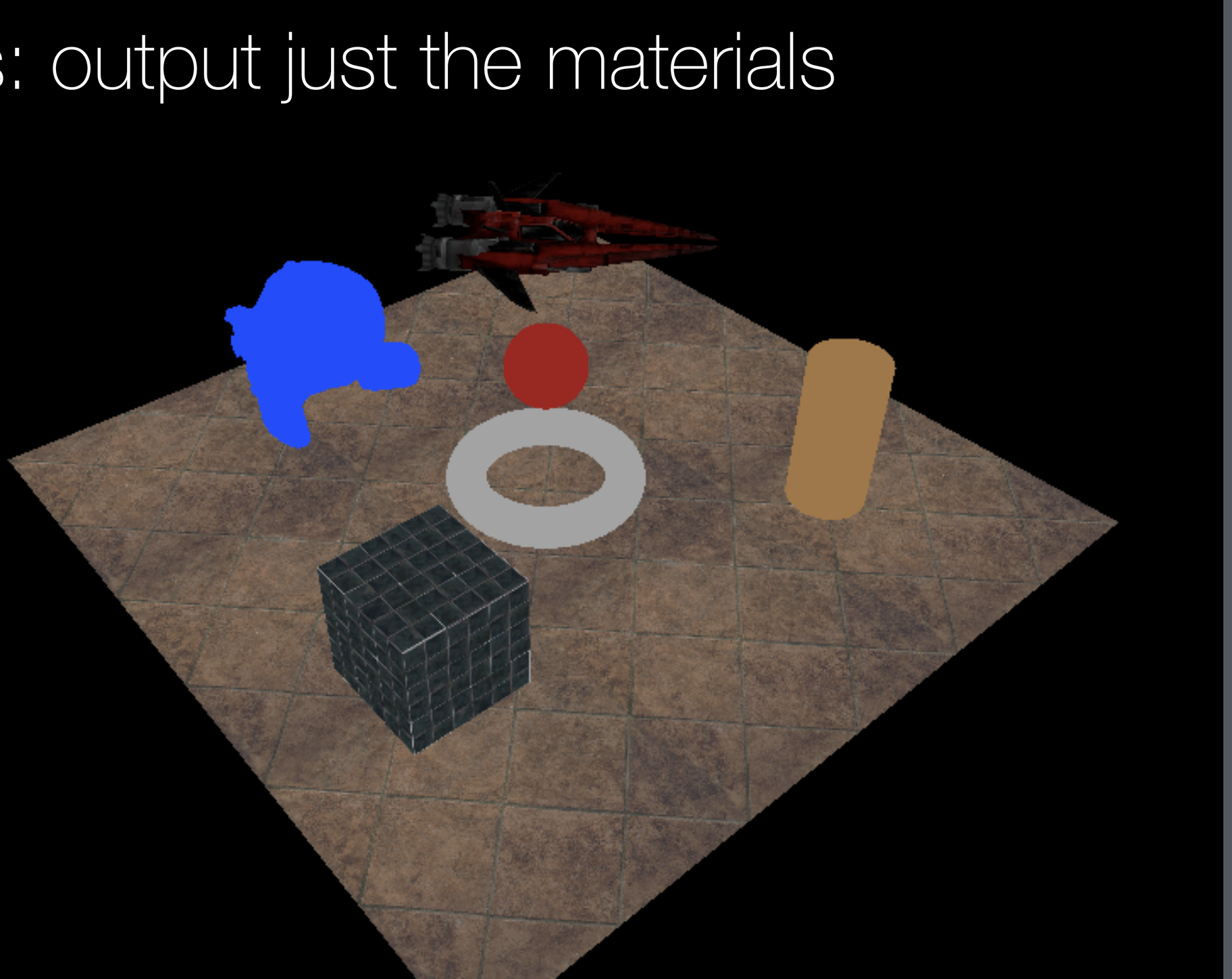
Post-processing pass (optional, can also be used with fwd. rendering)

- 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_FragColor = material properties of f
         if (depth of f < depthbuffer[x, y])
            gbuffer[x, y] = gl_FragColor
            depthbuffer[x, y] = depth of f
         end if

      end for
   end for
  end for
```

First pass: output just the materials

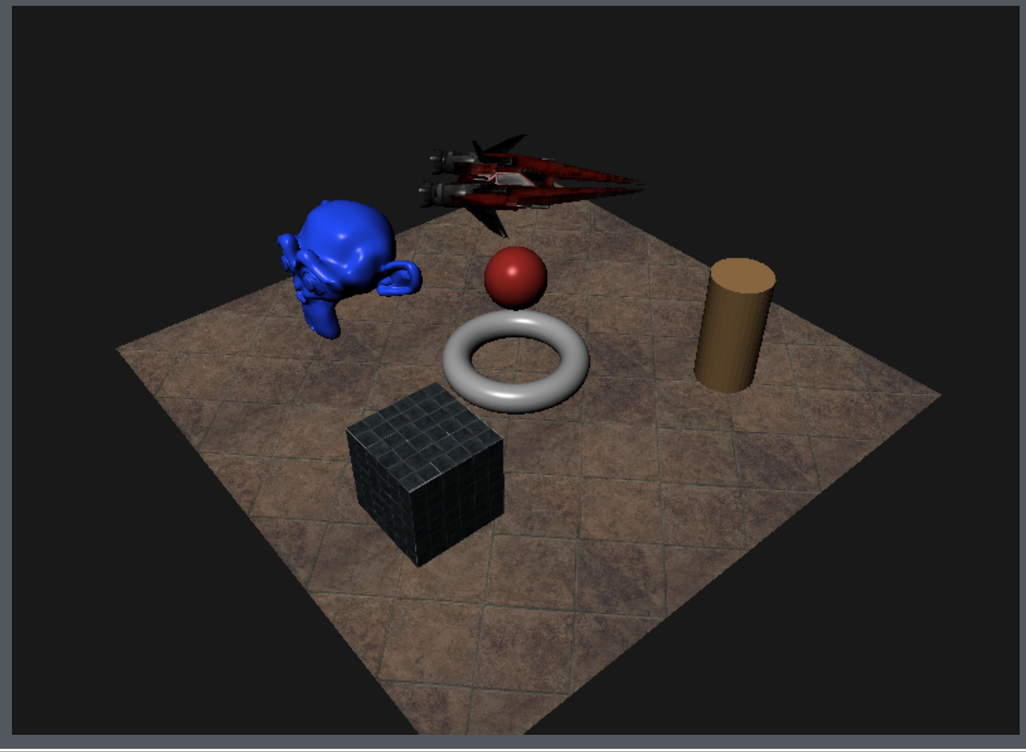


Deferred Shading Step 2

```
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_FragColor = material properties of f
         if (depth of f < depthbuffer[x, y])
            gbuffer[x, y] = gl_FragColor
            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
```

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:

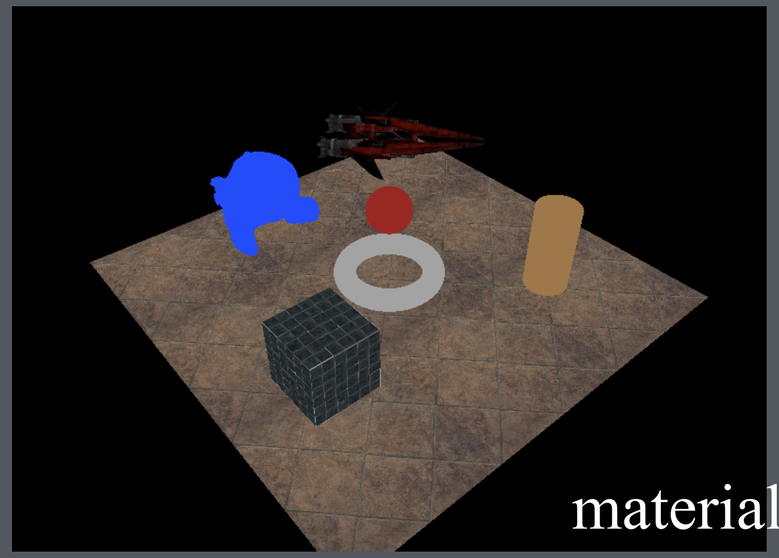
- Blinn-Phong: kd, ks, n
- Microfacet: kd, ks, alpha
- etc.

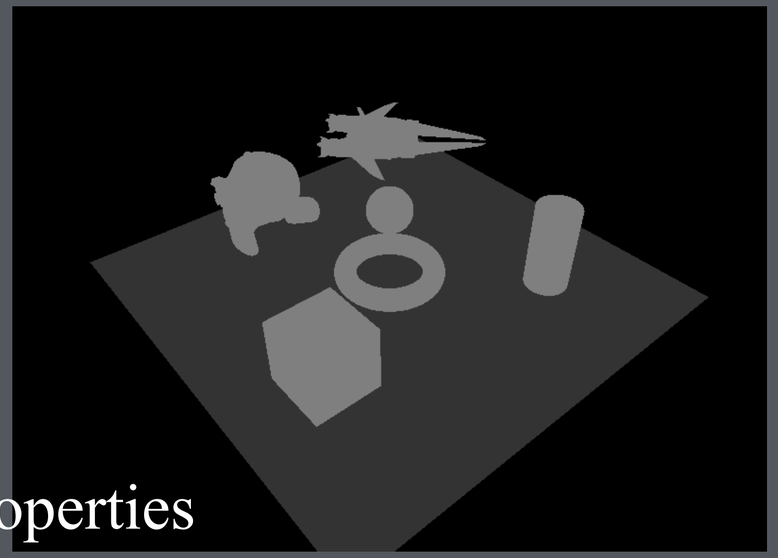
Also need fragment position and surface normal

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
```

G-buffer: multiple textures

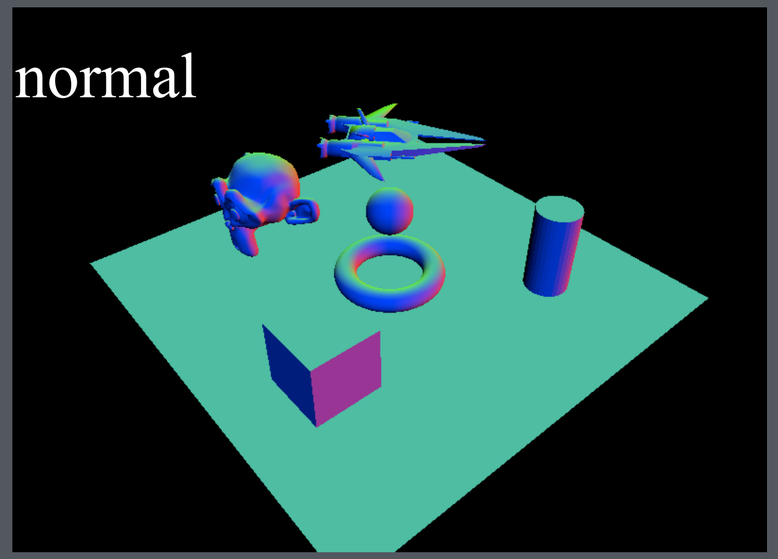




material properties

position

normal



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:

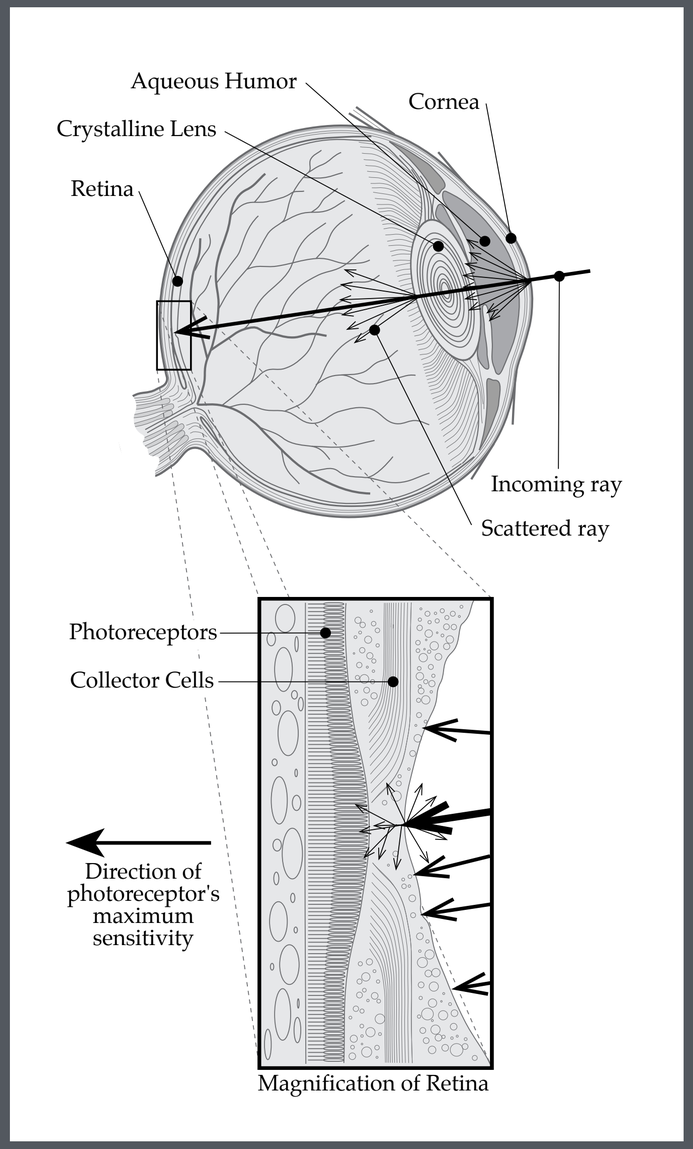
- screen-space ambient occlusion
- silhouette detection for artistic rendering

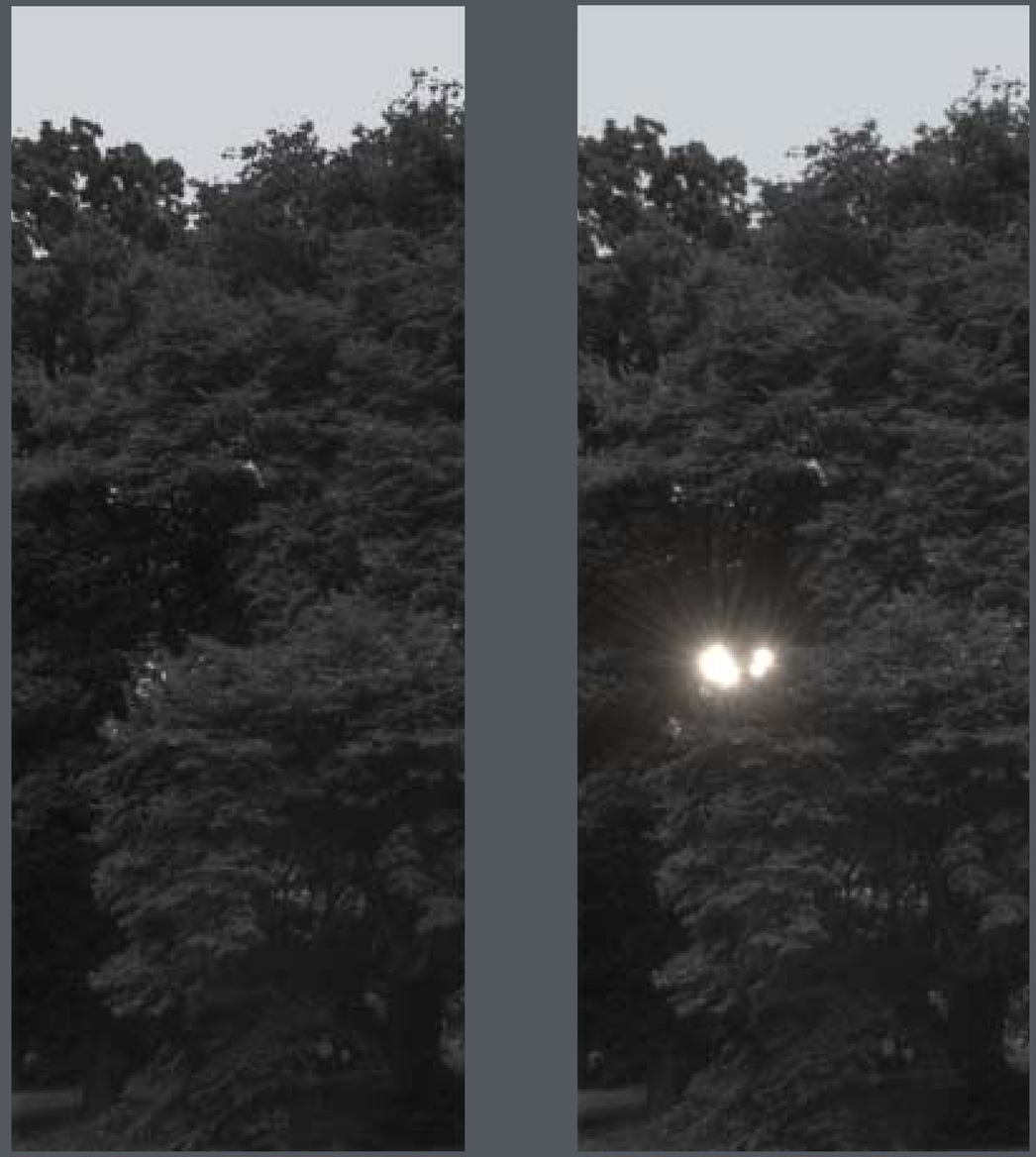
Amient occlusion

Bloom

Journey—Thatgamecompany (2012)

Modeling flare in the eye





Spencer et al. SIGGRAPH 1995

Modeling flare in a camera lens

Hullin et al. SIGGRAPH 2011

Color grading

The Matrix (Warner Bros.); comparison: Digital Media Academy

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

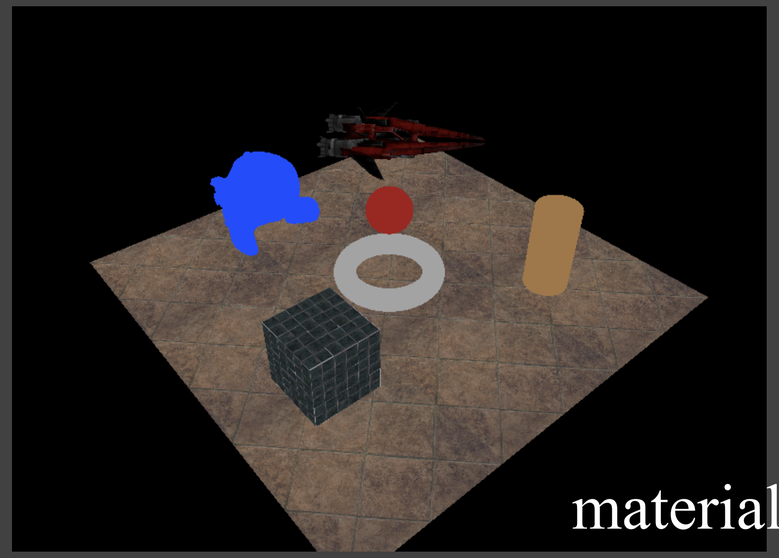
Cons:

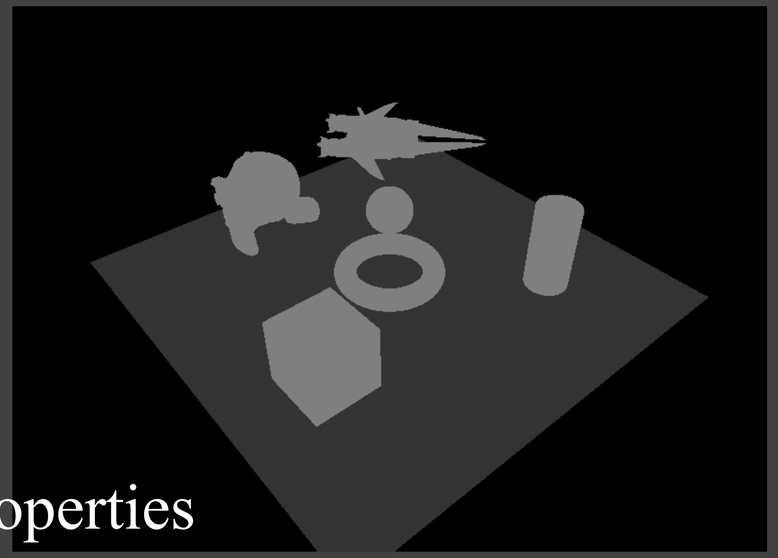
- transparency (only get one fragment per pixel)
- antialiasing (multisample AA not easy to adapt)

Engines (now)

- Cry Engine / Amazon Lumberyard
- Unreal Engine 3
- Unity 5

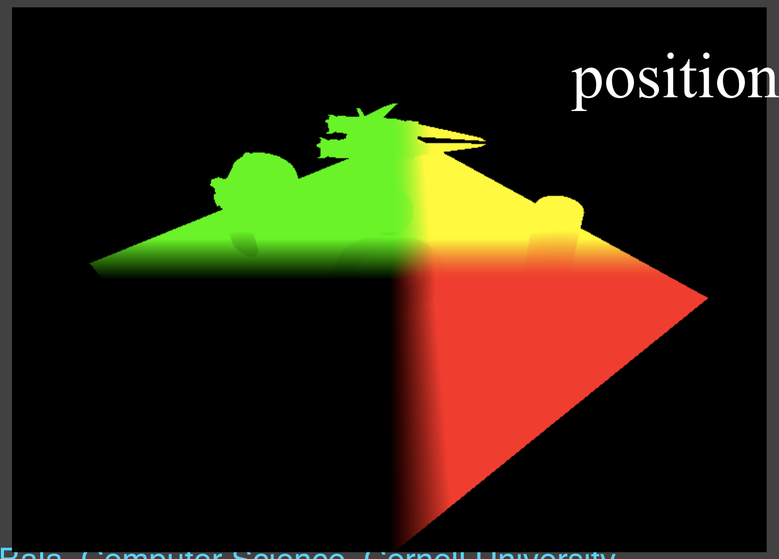
How to Fill the Buffers?



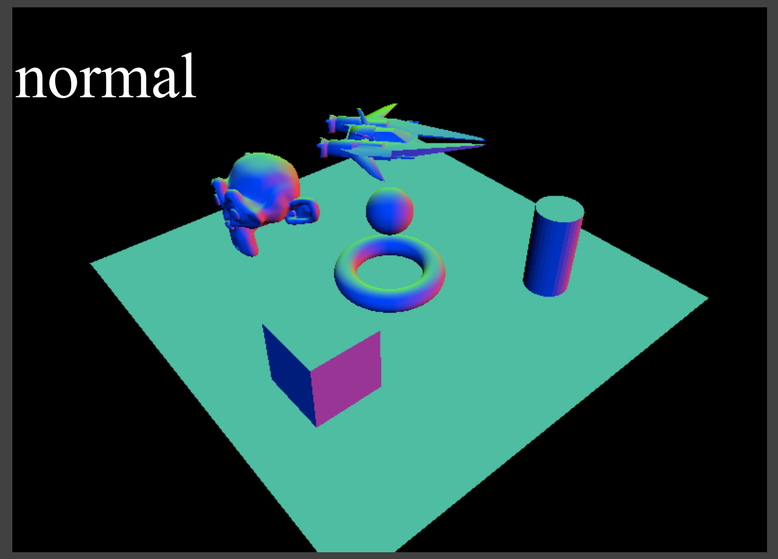


material properties

position



normal



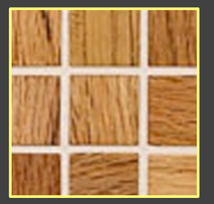
Single Render Target

- Requires N passes through scene:
  - glClear(): writes every pixel, is fairly expensive
  - Make hardware transform all the vertices again
  - Expensive if you are:
    - animating in your vertex shader
    - have subdivision surfaces generating lots of polygons
- Requires N different trivial shaders to output each variable you need (annoying)

Single Render Target

glClear();
renderScene(textureShader);

```
gl_FragColor = someTexture;
```

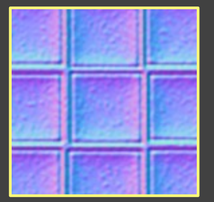


glClear();
renderScene(yellowShader);

```
gl_FragColor = yellow;
```

glClear();
renderScene(normalShader);

```
gl_FragColor = normal;
```



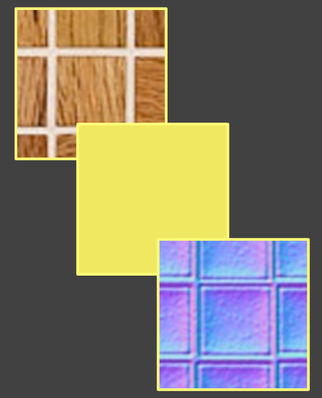
Multiple Render Targets

- Requires single pass through scene:
  - Single glClear() call
  - Single transformation of geometry
  - Single shader to output everything
- Fragment shader outputs multiple colors
  - Each to a different color attachment of a framebuffer object
- Different cards/drivers support different # of render targets
  - We use 4, which is widely supported

Multiple Render Targets

glClear();
renderScene(everythingShader);

```
gl_FragData[0] = someTexture;
gl_FragData[1] = yellow;
gl_FragData[2] = normal;
```



- Easier and more efficient than multiple passes
- Many “multi-pass” techniques in old papers can be done in a single pass now by using MRTs

Limitations of Deferred Shading

- Each pixel in the g-buffer can only store material and surface info for a single surface.
  - So, blending/transparency is difficult
  - Also, antialiasing is difficult
- For transparency: a “hybrid” renderer
  - Uses 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

Hybrid Rendering

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

Translucency cont.

- Hybrid renderer is the most practical solution.
- Can also do “depth peeling”:
  - Render and shade each layer of translucency using deferred shading
  - Sort of like a “deep” g-buffer
  - Gets very expensive very quickly

- Will probably implement a hybrid renderer to support translucent particle systems later on

Antialiasing

Single shading sample per pixel

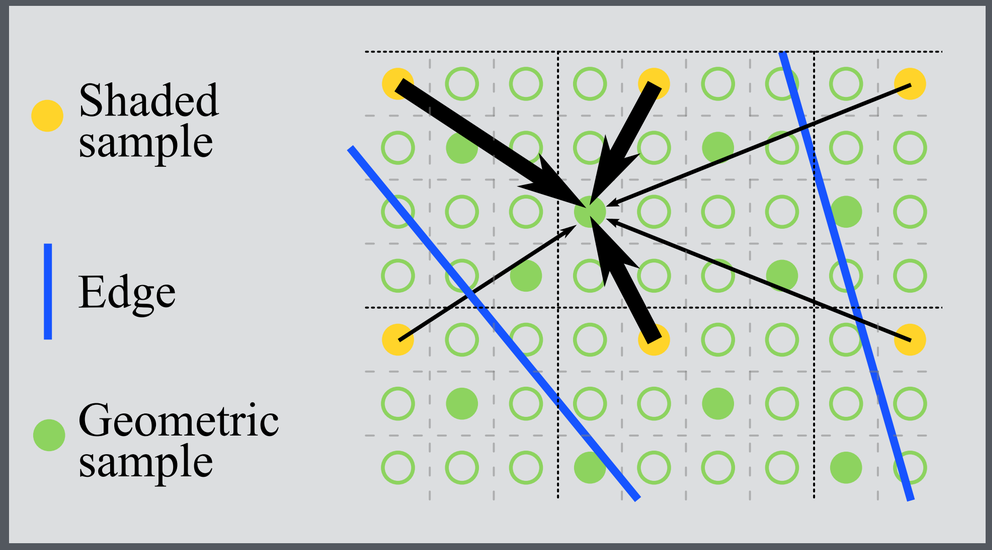
Reconstruct by blending nearby samples

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

MLAA

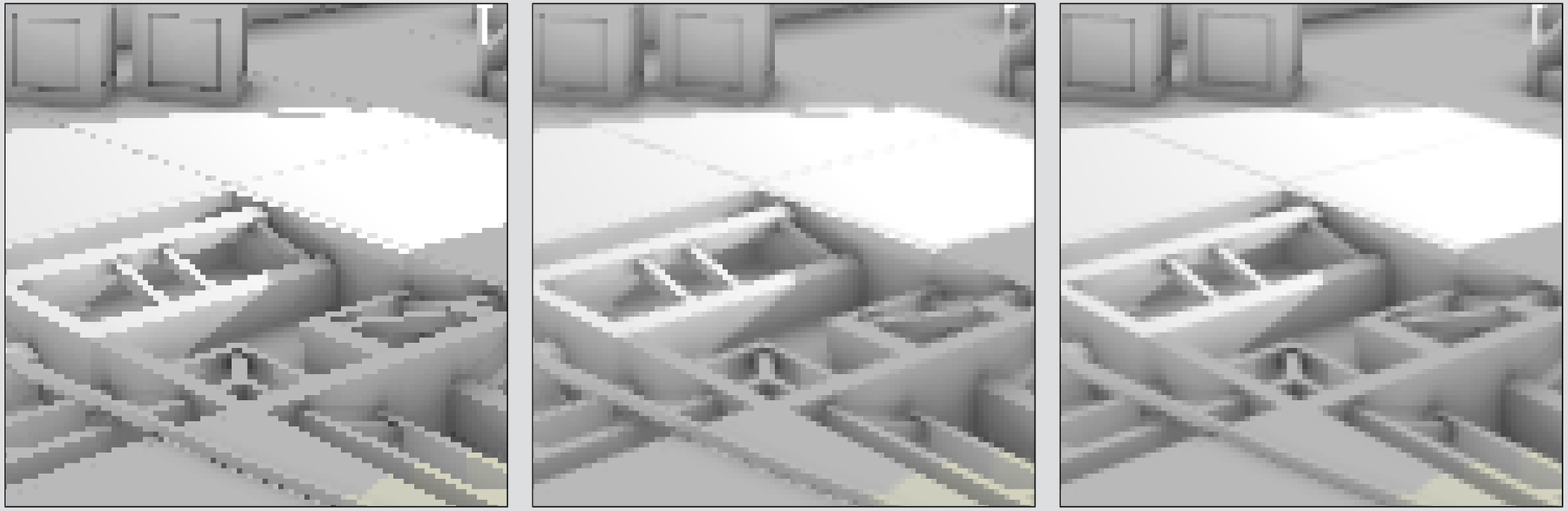
[Reshetov 09]

SRAA



← *Similar Time* →

← *Similar Quality* →



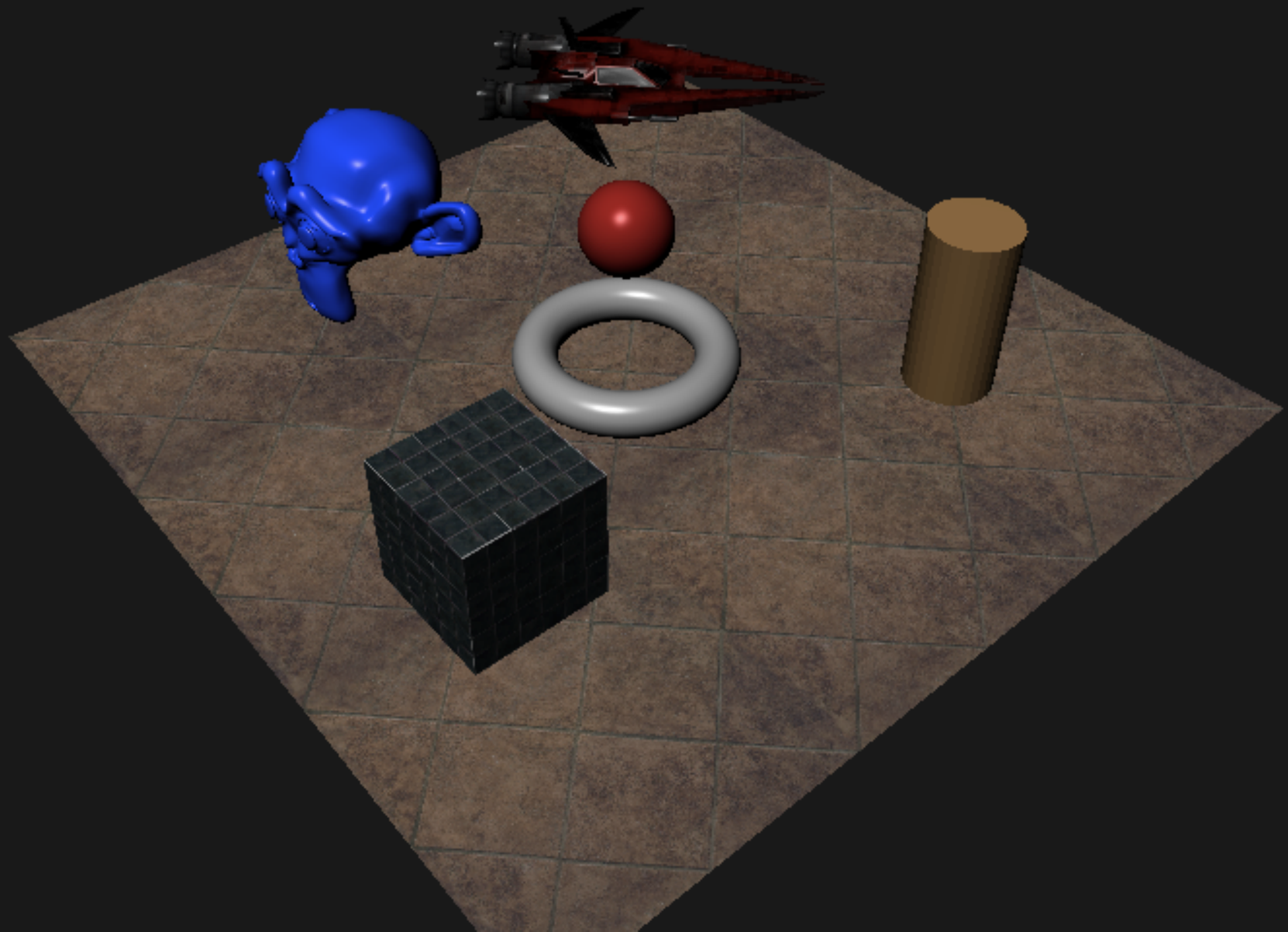
(a) 1× Shading + Box (poor, fast)

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

(c) 16× Shading + Box (good, slow)

[Chajdas et al. 11]

Output: the shaded scene

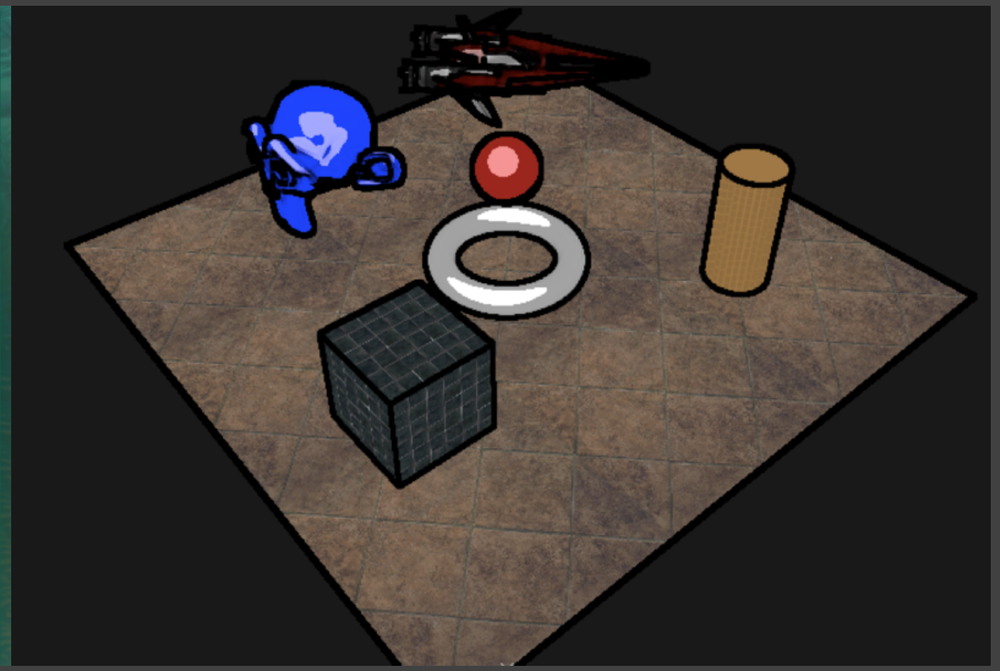


Drawbacks of Forward Shading

- If shade (f) is very expensive
  - e.g., many lights, shadow maps, complex shaders
  - overdraw by closer geometry wastes work on each fragment

Drawbacks of Forward Shading

- Many other complex effects: Image processing effects
  - tonemapping, screen-space ambient occlusion, bloom, toon shaders etc., are very expensive
    - Must read rendered image back from the framebuffer, or do entire pass through the scene geometry to render it to a FBO (frame buffer object)



Overdraw: Real Example

(Battlefield 3)

Deferred Shading Step 1

Code structure is nearly the same as forward shading, with one key difference:

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

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

        end for
     end for
  end for
```