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

            \texttt{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
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
Multi-pass rendering

An old idea

- going back at least to the 80s
- render something, store it, use the result to render something else
- used for dynamic reflections, shadows, etc.

Deferred rendering is a particular type of multi-pass approach

- store data in the image-space pixel grid
- store quantities that would be intermediate results in forward rendering
- use stored values in later pass(es) to compute final shading
- later passes can work on only some pixels, and can access data from other pixels
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

One improvement: $\text{shade (f)}$ 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.

```plaintext
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:

```plaintext
{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

- move loop over lights out of the shader: do a shading pass per light
- 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
By differentiating the depth buffer, can locate silhouettes and creases
Amient occlusion
Denoising

[Mara et al. HPG 2017]
After pixel values are computed, often more processing is desired

- effects that depend only on pixel values and depth (not normals, texture, etc.) are the same in forward or deferred mode
- pointwise color transforms: color grading, tone processing
- convolution filters: lens flare and bloom
- depth-dependent effects: atmospheric haze
Color grading

*The Matrix* (Warner Bros.); comparison: Digital Media Academy
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
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])
Processing textures for the Edgar model: (a) – original aliased image; (d) – antialiased image, processed with MLAA; (b,c) – enlarged regions of left and right images; (e,f,g) – visualization of pixels processed with MLAA at different zoom levels. Pixels belonging to horizontal shapes are marked with green, vertical – with red color. Pixels, which are included into horizontal and vertical shapes simultaneously, are shown as blue. Note that aliased pixels (top left) were unintentionally blurred when this paper was created (both in electronic and paper form).
Subpixel Reconstruction Antialiasing combines single-pixel shading with subpixel visibility to create antialiased images. It applies a joint bilateral filter inside each pixel based on subpixel geometric samples in a Latin square. Scene from grid. It applies a joint bilateral filter inside each pixel based on subpixel geometric samples in a Latin square. Notice that due to the fixed radius of the filter support, a variable support range. For instance a triangle kernel, could be employed for reconstruction strategy. The key part is to sample the shading at close subpixel resolution. See Section 3.4 for details.

SRAA benefits shading-bound applications. For example, our implementation resembles Morphological Antialiasing (MLAA), but with resolution depth and normal buffers, so it can be incorporated into existing shading renderers, which cannot use multisample antialiasing. SRAA operates as a post-process on a rendered image with subpixel reconstruction antialiasing. While these methods also achieve low shading rates, they are far cost in the resolution – or perform Morphological Antialiasing (MLAA) [Reshetov 2009], a sort of heuristic “smart blur” of the geometry in screen space to perform high-quality antialiasing. Typically, the filter radius is extremely narrow to avoid blurring and simplifying much computation. We have found that using only depth has a minor quality impact, and seems like a good trade-off for games; see Figure 4 for a comparison.

Deferred lighting, programs tend to either super sample – at linear cost in the resolution – or perform Morphological Antialiasing (MSAA) [Nichols et al. 2010]. Deferred lighting uses screen-space multiresolution gather methods to separate the computation of the shading of triangles from the computation of how many samples they cover. Deferred lighting uses MSAA resolves edges by computing distance. For the position change, we can estimate the distance to the closest plane that intersects the triangle. The plane distance metric caps the small insets on the boxes slightly better. This reduces the total number of G-buffer loads and allows us to easily increase the importance of the bilateral filter. We used the same data, SRAA creates high-resolution shading data suitable for filtering back down to screen resolution.

Our algorithm allows several performance/quality trade-offs by changing how the filter weights are computed. We can estimate the normal change term \( \mathbf{n} \) and a point \( \mathbf{p} \) at each subpixel sample. \( \mathbf{s} \) is the target sample and \( s \) is the normal change term. Thus SRAA would prove performance.

Notice that due to the fixed radius of the filter support, a variable support range. For instance a triangle kernel, could be employed for reconstruction algorithms up to large scenes. MSAA resolves edges by finding the closest plane with normal \( \mathbf{n} \), and a point \( \mathbf{p} \) and a point \( \mathbf{t} \) and a target sample \( s \) and a target sample \( s \). The plane distance metric caps the small insets on the boxes slightly better. This reduces the total number of G-buffer loads and allows us to easily increase the importance of the bilateral filter. We used the same data, SRAA creates high-resolution shading data suitable for filtering back down to screen resolution.

Typically, the filter radius is extremely narrow to avoid blurring and simplifying much computation. We have found that using only depth has a minor quality impact, and seems like a good trade-off for games; see Figure 4 for a comparison.

Figure 1: Subpixel Reconstruction Antialiasing (SRAA) combines single-pixel shading with subpixel visibility to create antialiased images.
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

- Renderbuffer:
  - Can only be used by attaching to an FBO

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

- FBO:
  - DEPTH_ATTACHMENT
  - STENCIL_ATTACHMENT
  - COLOR_ATTACHMENT0
  - COLOR_ATTACHMENT1

All these images have to match in size!