Deferred Shading

“Never do today what can be put off until tomorrow. Delay may give clearer light as to what is best to be done.” -Aaron Burr, maybe

Contents

1. Why use deferred shading?
   a. Forward Rendering benefits and drawbacks
   b. Multi-pass Forward Rendering benefits and drawbacks
   c. Deferred Shading benefits and drawbacks
2. How to implement a Deferred Shading system
   a. Geometry Stage
   b. Lighting Stage
   c. Addressing drawbacks of Deferred Shading
3. Optional Material - Additional Rendering systems

Forward Rendering

- Forward Rendering, also know as Direct Rendering, is the technique you have likely been most exposed to at this time.
- A simple Forward Renderer will render out final pixel colors by going through the following steps:

1. For each object to render
    a. Process each vertex in object (Vertex Shader)
        i. For each pixel covered by geometry of processed vertices (Pixel Shader)
            1. For each light that would affect the pixel
                a. Apply lighting algorithm

Forward Rendering

- Main benefit over other techniques we will discuss
  - Geometry is typically transformed only once
- Drawbacks
  - Complex shaders
    - Have to support a variable number, of variable types of lights
    - Differing types of materials and effects may need support requiring re-implementing shaders with additional permutations of effects supported
      - Normal Mapping
      - Skinned Animation
      - Normal Mapping + Skinned Animation
      - Remember, branching in shaders should typically be avoided where possible

Forward Rendering

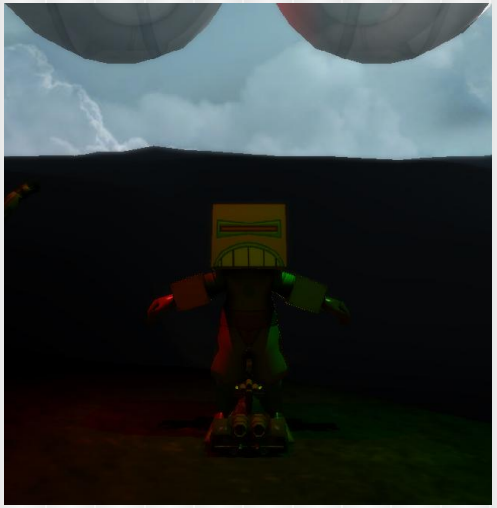
- Drawbacks continued...
  - Hardware limit on number of lights
    - We have a fixed number of registers we can use to supply data to shaders, such as light properties, resulting a fixed max number of lights we can support, typically around 8
    - This is the main drawback that typically causes us to not use forward rendering
  - Shadow Maps for all lights must be loaded
    - This can greatly increase memory footprint
    - We have a hardware limit on the number of different textures we can access in a given pass, which can limit how many shadow casting lights we support
  - High cost for overdraw

Multi-pass Forward Rendering

- For each light in the scene
  - For each mesh in range of the light
    - Render mesh, with just the current light applied
    - Additive blending will combine results into final image

Light one render +

Light two render =



Final Image

Multi-pass Forward Rendering

- Benefits
  - No hard limit on the number of lights. Each light is rendered separately, preventing us from running out of storage for lighting variables.
  - Shaders can be simpler. We will not need a single shader that can deal with unknown combinations of light types, instead we have one shader for each type needed.
  - Can/should/possible is easier to deal with which lights do or do not affect particular geometry. If the geometry being rendered is not in the bounds of the current light, don’t render.

Multi-pass Forward Rendering

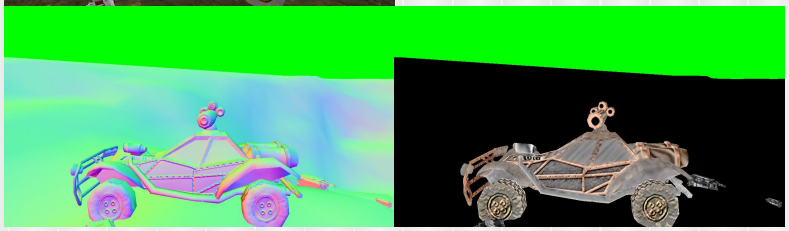
- Drawbacks
  - Repeated rendering work
    - Vertex shaders for geometry being rendered has to run for each light
    - Texture filtering work may be repeated
    - Possible increase to cache misses
  - Increased context switching
    - We either render all objects for each light, preventing us from rendering similar objects together, or we process each light for each object causing us to switching shaders and light parameters repeatedly.
  - High cost for overdraw
  - Worst case number of render passes
    - Number of objects * number of lights

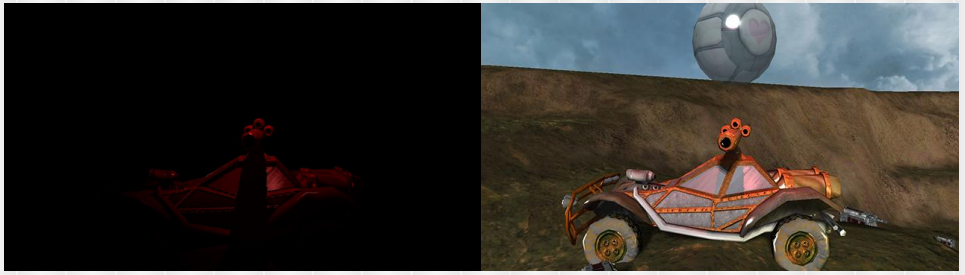
Deferred Shading

- Deferred Shading method
  - Broken into two stages, the geometry stage and the lighting stage
    - The geometry stage
      - CPU side we render the geometry we ultimately want to see
      - Shader side - instead of trying to find final colors, we output the information about the geometry we will need during the lighting stage.
        - We will need at least, **depth**, **normals** and **diffuse** color of the geometry, though we are not limited to just this data.
      - No lights during this stage
    - The lighting stage - next slide

Deferred Shading

- Deferred Shading method
  - Continued from previous slide...
    - The lighting stage
      - For each light, render geometry representing the volume the light would have an effect on.
        - For example, we would render a sphere for a point light
        - For each pixel covered by the light volume, use the data from the geometry stage to calculate final color information.
        - Additive blending is used to combine multiple lights





Deferred Shading Benefits

- Benefits
  - No hard limit on number of lights
    - Adding additional lights will have less impact on frame rate than other techniques
  - Reduced cost of over draw
    - Overdrawn will only happen during geometry stage, which is a relatively cheap stage compared to lighting
    - If overdraw is a bottleneck, it can be further reduced by rendering just depths before the rest of geometry stage
  - Simplified shaders
    - Special geometry techniques and lighting techniques are not combined
    - Each light type gets a separate shader
  - Geometry is typically transformed only once
  - Worst case number of passes = number object + number of lights

Deferred Shading Drawbacks

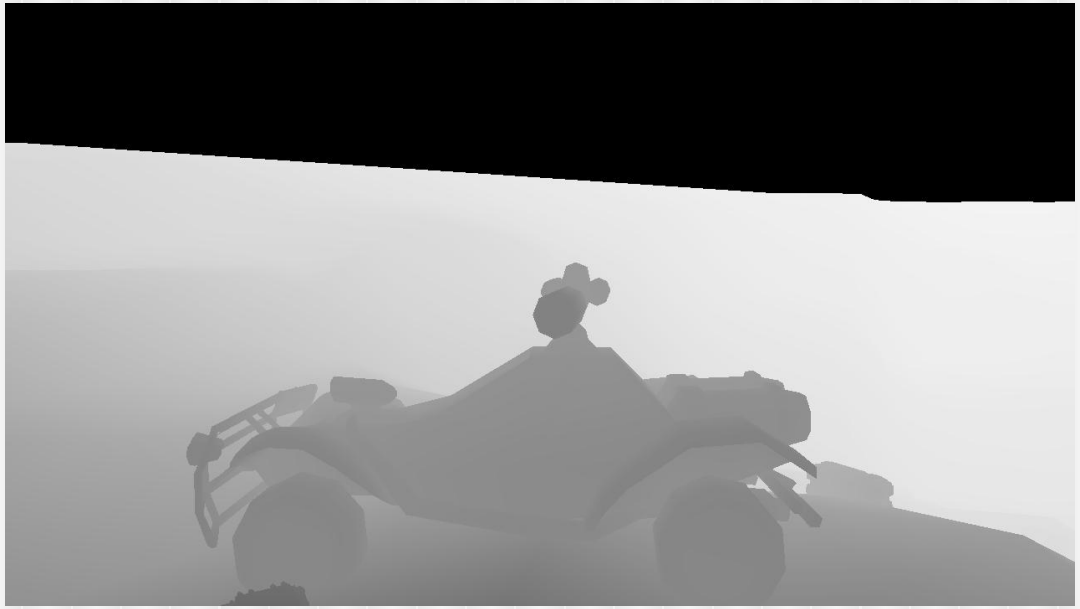
- Drawbacks
  - Memory footprint for geometry buffers
    - Four buffers at 32 bits per pixel, 1920x1080 resolution is around 31 MB of video ram
    - We may have more than 4 buffers and we may have higher than 32 bit pixel precision
  - Precision loss
    - How do we pack an XYZ float value into one 32 bit location?
  - Additional data packing and unpacking costs
    - Typically store data as RGBA colors with components ranging from 0 to 1
    - How do we store a normal with XYZ components that range from -1 to 1? A packing algorithm like color = (normal + 1) /2
  - Multi-sample anti-aliasing not well supported
  - Transparency does not work in deferred, but we will have it...

Geometry Stage

- During the geometry stage we will render out all geometry information we will need to later do lighting.
- Each type of data we write, depths, normals, diffuse, specular, will go to its own render target buffer.
- We will use Multiple Render Target support to render to all the geometry buffers in a single pass.
  - SV_TARGET0 = diffuse color
  - SV_TARGET1 = normal color
  - SV_TARGET2 = specular color
  - SV_TARGET3 = depth color
- The resulting buffers from this stage at commonly referred to as the “GBuffers”

Rendering Depths

- One of GBuffers we will need is the depth of each pixel
- The image below has been adjusted to make it easier to see. Depth values use most of their precision very near to the near clip plane, resulting in most depth buffers appearing nearly all white.
- Given a vertex and pixel shader, how do we render out depth values ranging from 0 to 1?



Rendering Depths pseudo-shader

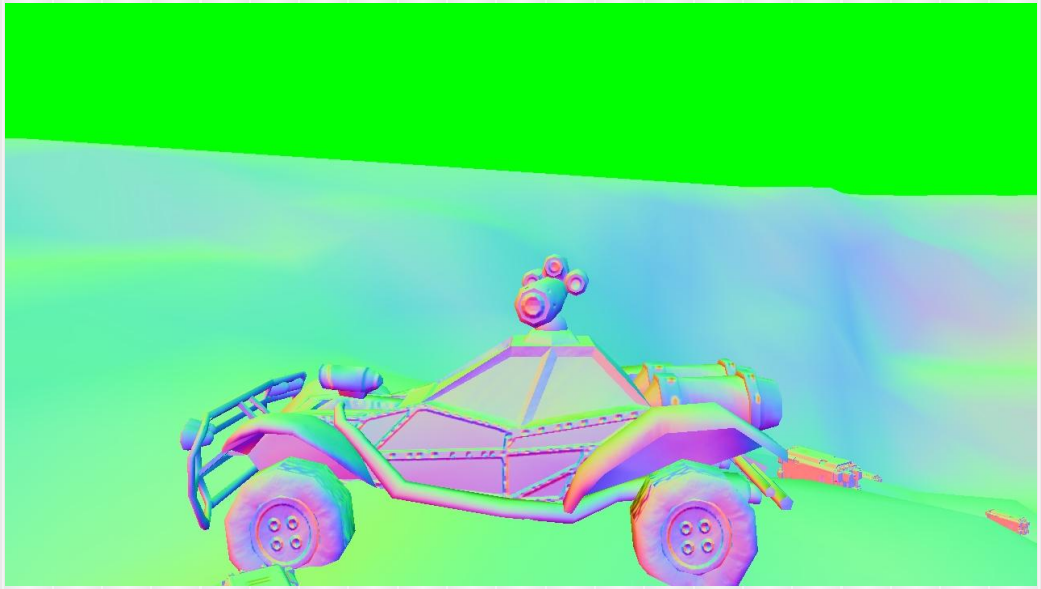
```
GBufferVertexOut main(float3 pos:POSITION0, …) // vertex shader
{
    GBufferVertexOut output;

    output.position = mul(float4(pos, 1.0), gMVP);
    …
    // output.position.z is depth ranging from Near clip too Far clip plane
    output.depthDiv = output.position.zw;
}
GBufferFragOut main(GBufferVertexOut input) // pixel shader
{
    GBufferFragOut output;
    …
    // output.depth is depth in clip space, ranging from 0 - 1
    output.depth = input.depthDiv.z / input.depthDiv.w;
}
```

Rendering Diffuse

- Under common single texture situations, we just sample the geometry's diffuse texture and return the result
- If we are using multiple textures, such as the terrain in our scene, we will need to flatten the results into a single color during this stage, or provide additional buffers for the other texture results. The first choice is the right one if at all possible.

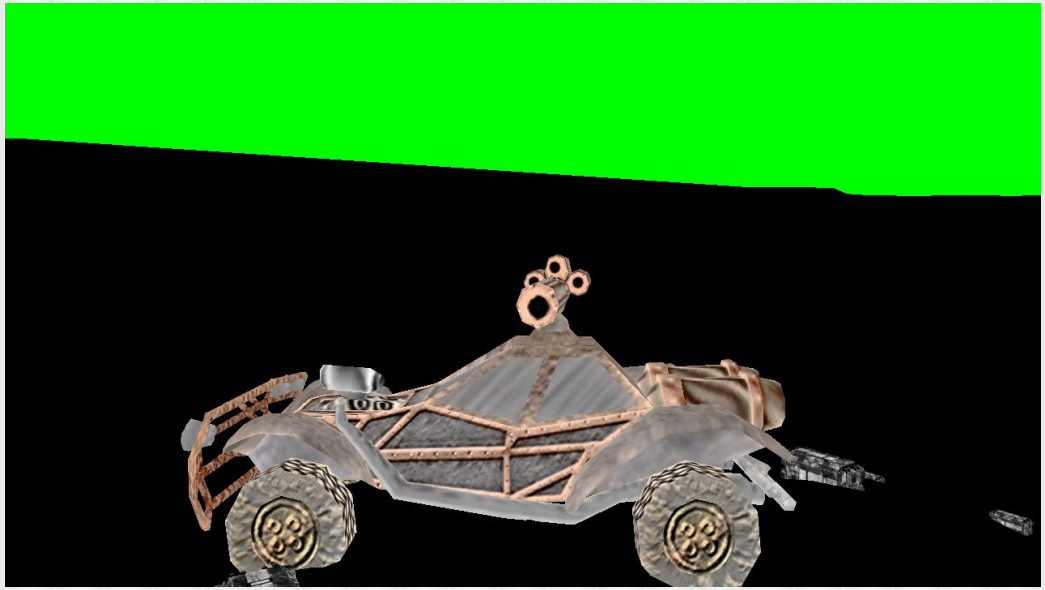
Rendering Normals

- We will need to pack the normal we receive from the vertex as a color
  - NormalColor = (Normal + 1) / 2;



Additional Material Properties

- Further buffers can be used to represent additional material properties
- We will use one of these buffers to represent how reflective our surface is, also known as its specular property
- Simply sample and return the specular texture



Multiple Render Targets

- The geometry being rendered to the multiple buffers will happen in a single pass, where all of the buffers are activated.
  - This is much better than rendering the geometry once for each buffer
- There are some requirements for using Multiple Render Targets (MRT) we need to keep in mind.
  - On DirectX 9 hardware we can have only 4 active at a time, on DirectX 10/11 we can have up to 8
  - All targets activated at the same time must have the same resolution and pixel depth, including the depth-stencil buffer
  - While they must have the same pixel depth, they do not have to have the same format.

Pixel Use with MRTs

- The pixel format used will vary with the implementation of the deferred shading system
- A simple approach would be to use the DXGI_FORMAT_R8G8B8A8_UNORM format for each of the colored buffers and, DXGI_FORMAT_R32_UNORM for depths
  - 8 bit for Normal.X, 8 for Normal.Y, 8 for Normal.Z, 8 null
  - 8 bit for Diffuse.R, 8 for Diffuse.G, 8 for Diffuse.B, 8 null
  - 8 bit for Spec.R, 8 for Spec.G, 8 for Spec.B, 8 SpecPower
  - 32 bit depth buffer
- Notice the rather extreme loss of precision, particularly with respect to the normals where a 32 bit value is being written to an 8 bit value.

Normals precision

- One common technique to improving the precision of normals is to use the DXGI_FORMAT_R16G16_UNORM format and only store the X and Y components
- The Z component can be rebuilt from the X and Y components
  - Example, Z = sqrt(X ^ 2 - Y ^ 2)
  - Square Root always has two possible answers
    - sqrt(4) == 2 && sqrt(4) == -2
  - So how do we pick if the Z should positive or negative?
    - One option is to pack a sign bit in another buffer somewhere, maybe like specularColor.A. This adds a strange complexity.
    - More common option is to move all lighting to happen in view space, as though the camera is the origin. A negative Z would then represent a backface, which we would not be rendering.
      - This is more optimized but will require a fair amount of refactoring.

FSGDEngine

- At the time of this writing we use:
  - DXGI_FORMAT_R8G8B8A8_UNORM for diffuse color
    - We will end up storing Ambient Occlusion values in the A channel
  - DXGI_FORMAT_R10G10B10A2_UNORM for normals
    - Decent enough precision for our use, the A channel is wasted
  - DXGI_FORMAT_R10G10B10A2_UNORM for specular color
    - We can support full color specular maps, though this is rarely used, could be changed to grey scale to pack more data into this buffer if needed.
  - DXGI_FORMAT_R32_UNORM for depths
    - This is totally for teaching. In practice we do not need a color buffer for depths, we would instead reference the actual Depth Buffer for the same depth values.

Geometry Stage complete

- These 4 buffers will contain all the information we will need to perform lighting, later.
  - Notice that we did not store the world space location of the fragment. Storing the position would have even worse precision issues than we have with normals as the numbers will have a much larger range of values.
    - We will have enough information to rebuild position during the lighting phase.
  - We call these buffers the GBuffers, and their data will be used for lighting as well as many other visual effects.
    - This data will be useful for post processing effects like Depth of Field, Fog, and Edge Detection which will be covered in another lecture.

Lighting Stage

- General Approach
  - G-Buffer now has all the data we will need during lighting stage
  - For each light in view, we need to calculate lighting results for all pixels that would be affected by the given light.
  - We additively blend the results of each rendered light together, much the same as we would do using a multi-



Rendering a Directional Light

- Directional lights should affect all pixels in view.
- Render a full screen quad
  - For each fragment covered by the quad (all of them)
    - Sample depth, diffuse color, and specular color
    - Sample normal color, and convert to normal range (unpack)
    - Calculate fragment position (Explained in next slide)
      - Perform standard lighting calculations. Other than where we get the data from such as normals and diffuse colors, nothing in our lighting model will be different than in a forward rendering system.
- We will end up repeating these same steps for point and spot lights, but rendering a sphere or cone respectively instead of a full screen quad.

Calculating Fragment position

- The formula is the same for each of the lights, but typically makes the most sense while thinking about rendering the full screen quad for the directional light.

1. Pass vertex position to fragment shader. This position will represent the light volume, not the actually geometry so it should not be used directly.
2. The XY and components of this passed position will be correct for the final fragment position in view space. Think of the origin being the camera, the interpolated XY of the light volume will match the XY of any geometry in that fragment.
3. For the Z component we will use a sample from our Depth buffer.
    a. Example, float4 viewPos(input.Pos.XY, depth, 1)

Continued...

Calculating Fragment Position

4. Typically we will convert the found view space position to world space.
   a. If we were doing the optimization in which we move normals to view space, we would be able to skip this step.
5. To convert from view space to world space, multiply the position by the inverse view-projection.
   a. float4 worldPos = mul(viewPos, gInvViewProj);
6. If the passed in view position is based off an object that had a perspective matrix applied, we will get a clip space value back, which needs one more conversion.
   a. worldPos /= worldPos.w
   b. This is needed for the point and spot light but not the directional light, but should not break the directional light if done.

Inverse

- They are expensive, so don’t do them.
  - Okay that is not possible, but avoid doing them where possible.
- The inverse we do to build a camera view matrix from its world is a fast inverse that only works for Ortho-Normalized matrices (XYZ axis are all normalized && XYZ axis are perpendicular to each other)
- We only inverse our camera’s view-projection once per frame, not for each vertex or each pixel.
- If you do a matrix inverse in a shader you are likely doing something wrong and other programmers will make fun of you. :)

Point and Spot light optimizations

- Point and spot lights will not typically affect all the pixels on the screen.
- If we can quickly limit the processing of these light types to only the fragments they will affect, adding additional lights to the scene will have very little effect on the frame rate.
  - Only when the range of two lights overlap will we start to get additional cost in their processing.
- So how do we limit the processing to match the volume of the light?

Point and Spot light optimizations

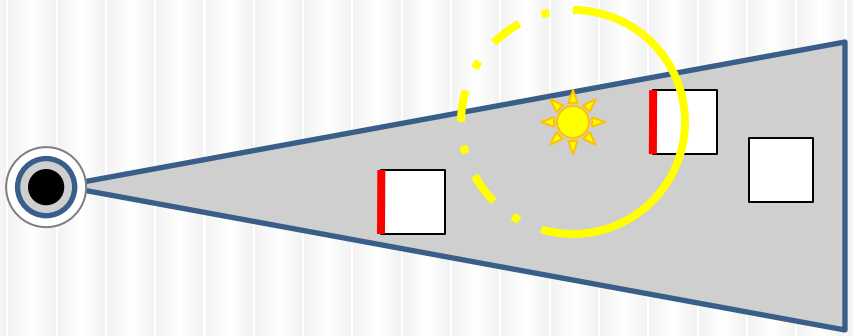
- Some techniques will render a full screen quad for all light types.
- These techniques will then use either scissor tests or stencil tests to limit the quad to only rendering fragments where the light volume actually covers.

- This is a 2D solution, it will still process fragments far in front of or behind the light that fall into the 2D volume…
- These full quad techniques are typically used with other rendering techniques that tile the screen into small pieces that get processed one at at time. We will do a 3D solution instead...

Point and Spot light optimizations

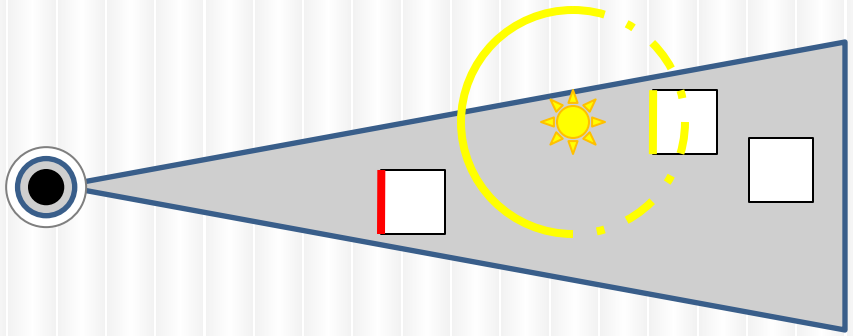
- 3D solution
  - Render volumes that match the shape of the light, like a sphere for a point light
  - This will (typically) be a three pass process

1. Render back faces of light volume, to stencil buffer only
   a. Set depth testing to pass for Greater
      i. This causes only the fragments of the sphere that are behind something to render
   b. Increment the stencil buffer where we render



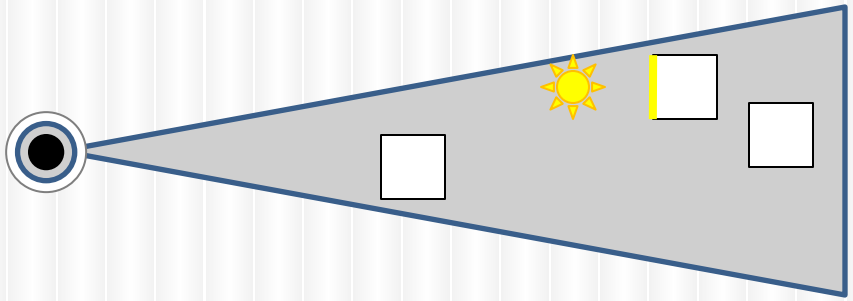
Point and Spot light optimizations

2. In the second pass we will render the volume again, with depth and stencil states reversed
   a. Set depth testing to pass for Less
      i. This causes only the fragments of the sphere that are **in front of** something to render
   b. Increment the stencil buffer where we render



Point and Spot light optimizations

3. In the third stage we will render our lights, limiting them to fragments that have been stenciled twice.
   a. The fragments that have been stenciled twice are behind the front faces and in front of the back faces, A.K.A in side of the volume.

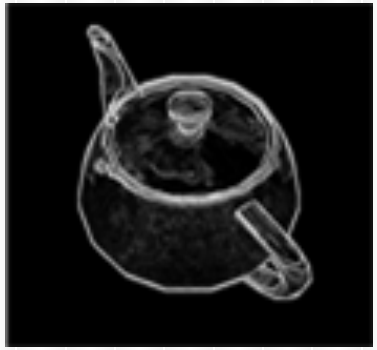


Addressing drawbacks of Deferred Shading

- Transparency is not handled well by a deferred shading system.
  - Do we blend the unlit colors of the geometry? How do we blend a normal?
- There are some rather complex techniques to make transparency work in a deferred shading system, typically using a stipple pattern.
  - Read white paper on Inferred Lighting for more on this approach
- A common, and far easier approach we will take is to render our transparent set using Forward rendering after our light volumes have rendered.
  - This will use a different lighting model, with a smaller number of max lights, but transparent objects are by definition harder to see, so we usually can get away with this.

Addressing drawbacks of Deferred Shading

- Another large issue we have with a deferred shading system is the lack of support for MSAA.
  - This is not the only path to anti-aliasing.
  - We could use a post-process edge detection shader to find the edges, which is where the aliasing will be.
  - We can then blur those regions.
  - This is a very dumbed down summary of what an FXAA (Fast Approximate Anti-Aliasing) effect does.
    - Download the FXAA demo and document from nVidia for more details.



Additional Materials

- Deferred Lighting, A.K.A. Light Pre-Pass
  - This is a similar system to deferred shading
    - Has three passes, instead of two, geometry, light, then geometry again
    - Supports MSAA, and requires less geometry data in GBuffers
    - http://www.realtimerendering.com/blog/deferred-lighting-approaches/
- Inferred Lighting, used at Volition for Saints Row series
  - This is also a similar system to deferred shading
    - Has three passes, geometry, lights, materials
    - Supports MSAA, and flexible material properties
    - White paper explains a technique for transparency rendering that would likely for any deferred model. Reasonably complex to implement.
    - White paper

Additional Materials

- Image-Based Anti-Aliasing
  - Fast Approximate Anti-Aliasing (FXAA)
    - This is a post-process anti-aliasing technique.
      - It works on a “picture” of your final scene
      - Works well with deferred shading model
      - nVidia Demo comes with source code and a fair explanation of how it works were you to try implementing the technique yourself.
      - https://developer.nvidia.com/nvidia-graphics-sdk-11-direct3d