6800 LEAGUES UNDER THE SEA

Deferred Shading

Mark Harris

The Challenge: Real-Time Lighting

- **Modern games use many lights on many objects covering many pixels**
  - **computationally expensive**
- **Three major options for real-time lighting**
  - **Single-pass, multi-light**
  - **Multi-pass, multi-light**
  - **Deferred Shading**
- **Each has associated trade-offs**

Comparison: Single-Pass Lighting

```
For Each Object:
  Render object, apply all lighting in one shader
```

- Hidden surfaces can cause wasted shading
- Hard to manage multi-light situations
  - Code generation can result in thousands of combinations for a single template shader
- Hard to integrate with shadows
  - Stencil = No Go
  - Shadow Maps = Easy to overflow VRAM

NVIDIA.

Comparison: Multipass Lighting

```
For Each Light:
  For Each Object Affected By Light:
     framebuffer += brdf( object, light )
```

- Hidden surfaces can cause wasted shading
- High Batch Count (1/object/light)
  - Even higher if shadow-casting
- Lots of repeated work each pass:
  - Vertex transform & setup
  - Anisotropic filtering

Comparison: Deferred Shading

```
For Each Object:
  Render lighting properties to "G-buffer"
For Each Light:
  framebuffer += brdf( G-buffer, light )
```

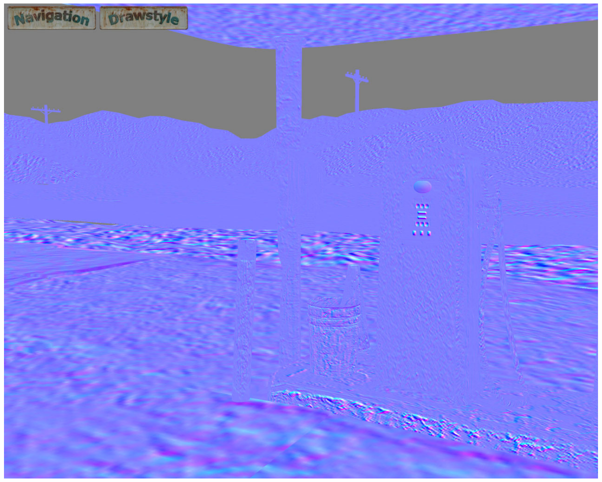
- **Greatly simplifies batching & engine management**
- **Easily integrates with popular shadow techniques**
- **"Perfect" O(1) depth complexity for lighting**
- **Lots of small lights ~ one big light**

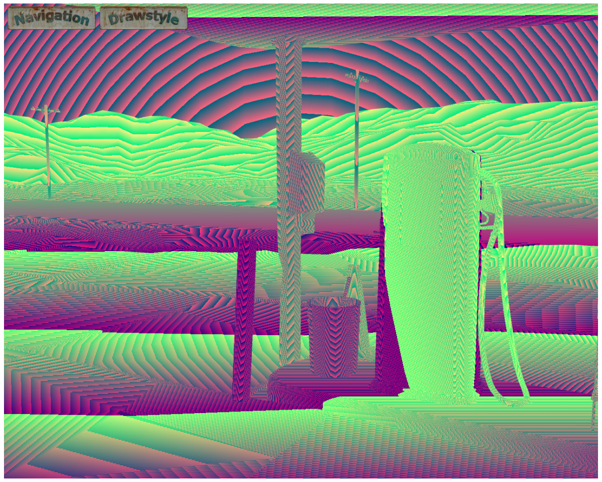
Deferred Shading: Not A New Idea!

- Deferred shading introduced by Michael Deering et al. at SIGGRAPH 1988
  - Their paper does not ever use the word “deferred”
  - PixelFlow used it (UNC / HP project)
- Just now becoming practical for games!

What is a G-Buffer?

- G-Buffer = All necessary per-pixel lighting terms
  - Normal
  - Position
  - Diffuse / Specular Albedo, other attributes
  - Limits lighting to a small number of parameters!





What You Need

- Deferred shading is best with high-end GPU features:
  - Floating-point textures: must store position
  - Multiple Render Targets (MRT): write all G-buffer attributes in a single pass
  - Floating-point blending: fast compositing

NVIDIA.

Attributes Pass

- Attributes written will depend on your shading
- Attributes needed
  - Position
  - Normal
  - Color
  - Others: specular/exponent map, emissive, light map, material ID, etc.

- Option: trade storage for computation
  - Store pos.z and compute xy from z + window.xy
  - Store normal.xy and compute $z=sqrt(1-x^2-y^2)$

MRT rules

- Up to 4 active render targets
- All must have the same number of bits
- You can mix RTs with different number of channels
- For example, this is OK:
  - RT0 = R32f
  - RT1 = G16R16f
  - RT2 = ARGB8
- This won't work:
  - RT0 = G16R16f
  - RT1 = A16R16G16B16f

Example MRT Layout

- **Three 16-bit Float MRTs**

RT1	Diffuse.r	Diffuse.g	Diffuse.b	Specular
RT0	Position.x	Position.y	Position.z	Emissive
RT2	Normal.x	Normal.y	Normal.z	Free

- **16-bit float is overkill for Diffuse reflectance…**
  - **But we don’t have a choice due to MRT rules**

Computing Lighting

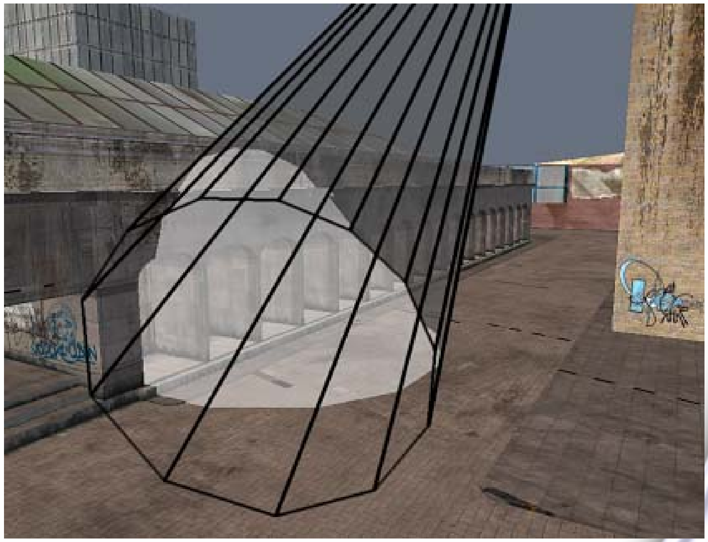
Render convex bounding geometry

- Spot Light = Cone
- Point Light = Sphere
- Directional Light = Quad or box

Read G-Buffer

Compute radiance

Blend into frame buffer



Courtesy of Shawn Hargreaves, GDC 2004

- Lots of optimizations possible
  - Clipping, occlusion query, Z-cull, stencil cull, etc.

Lighting Details

- **Blend contribution from each light into accumulation buffer**
  - **Keep diffuse and specular separate**

```
For each light:
  diffuse += diffuse(G-buff.N, L))
  specular += G-buff.spec *
                   specular(G-buff.N, G-buff.P, L)
```

- **A final full-screen pass modulates diffuse color:**

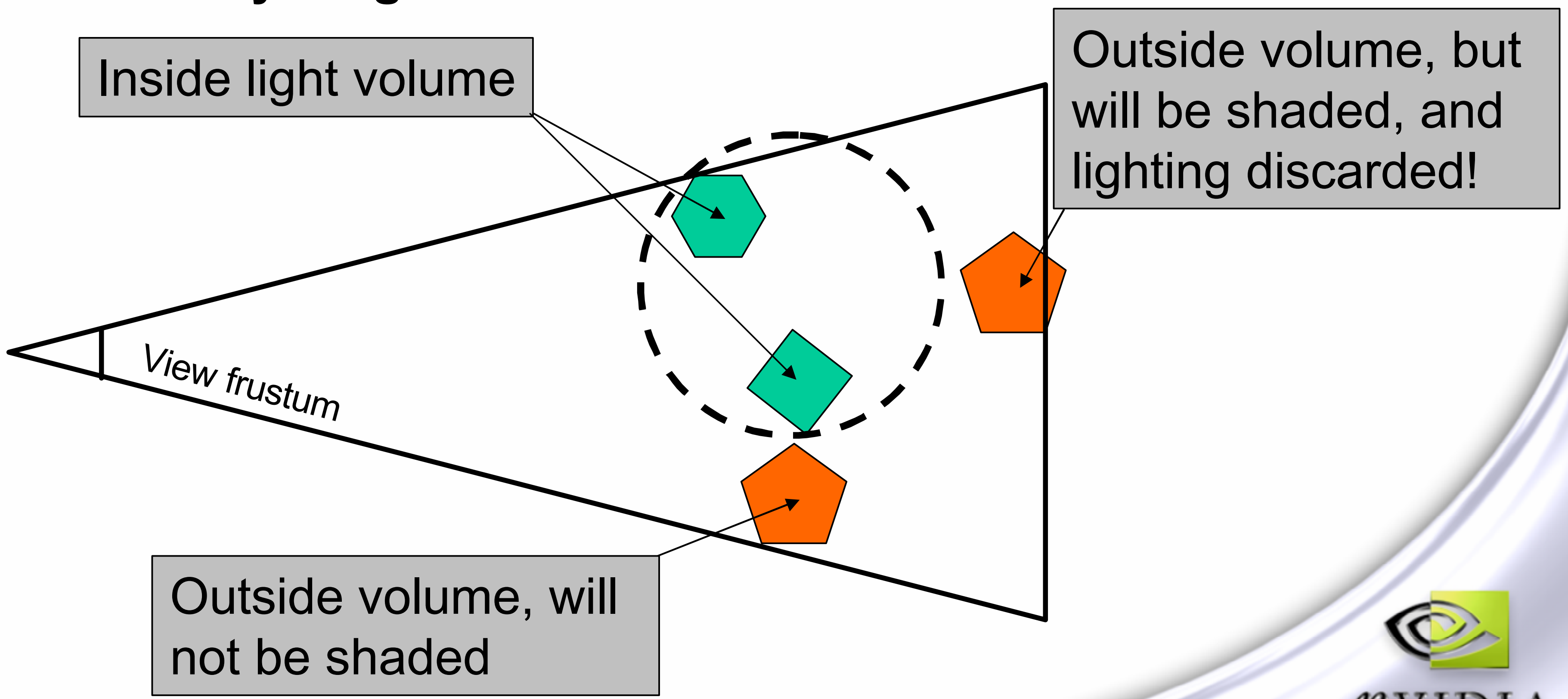
```
framebuffer = diffuse * G-buff.diffuse + specular
```

Options for accumulation buffer(s)

- Precision
  - 16-bit floating point enables HDR
  - Can use 8-bit for higher performance
    - Beware of saturation
- Channels
  - RGBA if monochrome specular is enough
  - 2 RGBA buffers if RGB diffuse and specular are both needed.
  - Small shader overhead for each RT written

Lighting Optimization

- Only want to shade surfaces inside light volume
  - Anything else is wasted work



Optimization: Stencil Cull

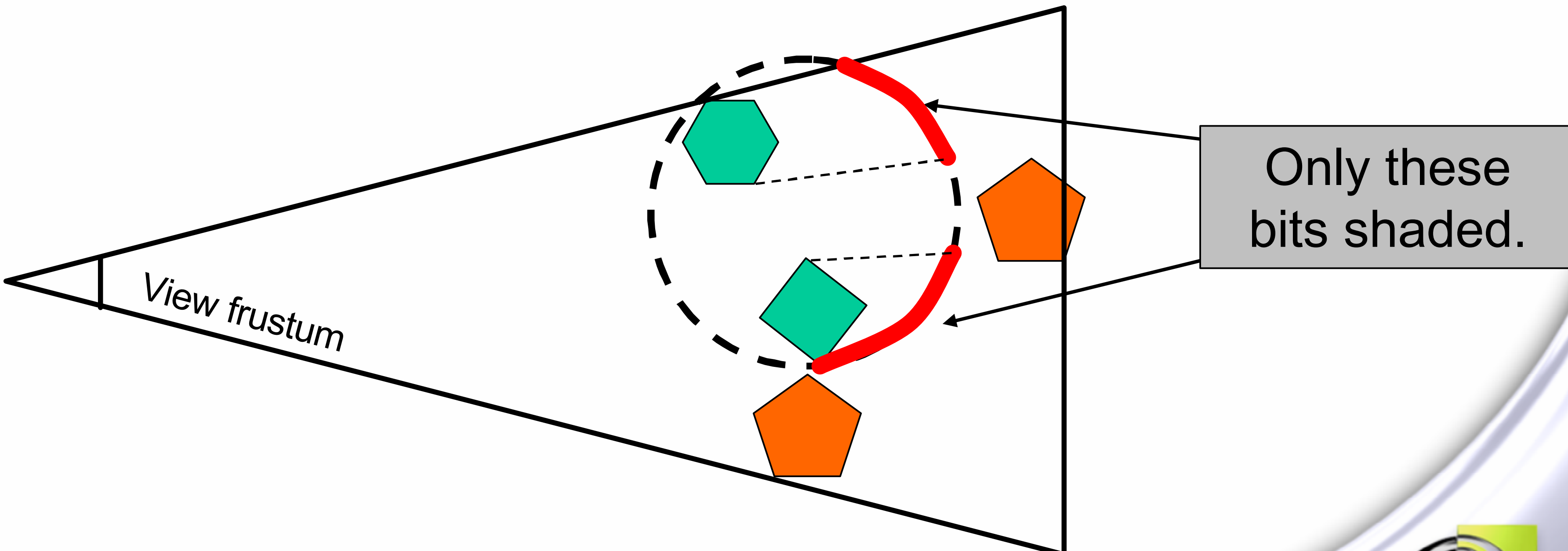
- Two pass algorithm, but first pass is very cheap
  - Rendering without color writes = 2x pixels per clock

1. Render light volume with color write disabled
   - Depth Func = LESS, Stencil Func = ALWAYS
   - Stencil Z-FAIL = REPLACE (with value X)
   - Rest of stencil ops set to KEEP
2. Render with lighting shader
   - Depth Func = ALWAY, Stencil Func = EQUAL, all ops = KEEP, Stencil Ref = X
   - Unlit pixels will be culled because stencil will not match the reference value

NVIDIA.

Setting up Stencil Buffer

- **Only regions that fail depth test represent objects within the light volume**



Shadows

- Shadow maps work very well with deferred shading
  - Work trivially for directional and spot lights
  - Point (omni) lights are trickier…
- Don't forget to use NVIDIA hardware shadow maps
  - Render to shadow map at 2x pixels per clock
  - Shadow depth comparison in hardware
  - 4 sample percentage closer filtering in hardware
  - Very fast high-quality shadows!
- May want to increase shadow bias based on pos.z
  - If using fp16 for G-buffer positions

Virtual Shadow Depth Cube Texture

- **Solution for point light shadows**
  - **Technique created by Will Newhall & Gary King**
- **Unrolls a shadow cube map into a 2D depth texture**
  - **Pixel shader computes ST and depth from XYZ**
  - **G16R16 cubemap efficiently maps XYZ->ST**
  - **Free bilinear filtering offsets extra per-pixel work**
- **More details in *ShaderX*^{3}**
  - **Charles River Media, October 2004**

Multiple Materials w/ Deferred Shading

- Deferred shading doesn't scale to multiple materials
  - Limited number of terms in G-buffer
  - Shader is tied to light source – 1 BRDF to rule them all
- Options:
  - Re-render light multiple times, 1 for each BRDF
    - Loses much of deferred shading's benefit
  - Store multiple BRDFs in light shader, choose per-pixel
    - Use that last free channel in G-buffer to store material ID
    - Reasonably coherent dynamic branching
    - Should work well on pixel shader 3.0 hardware

Transparency

- Deferred shading does not support transparency
  - Only shades nearest surfaces
- Just draw transparent objects last
  - Can use depth peeling
  - Blend into final image, sort back-to-front as always
  - Use “normal” shading / lighting
  - Make sure you use the same depth buffer as the rest
- Also draw particles and other blended effects last

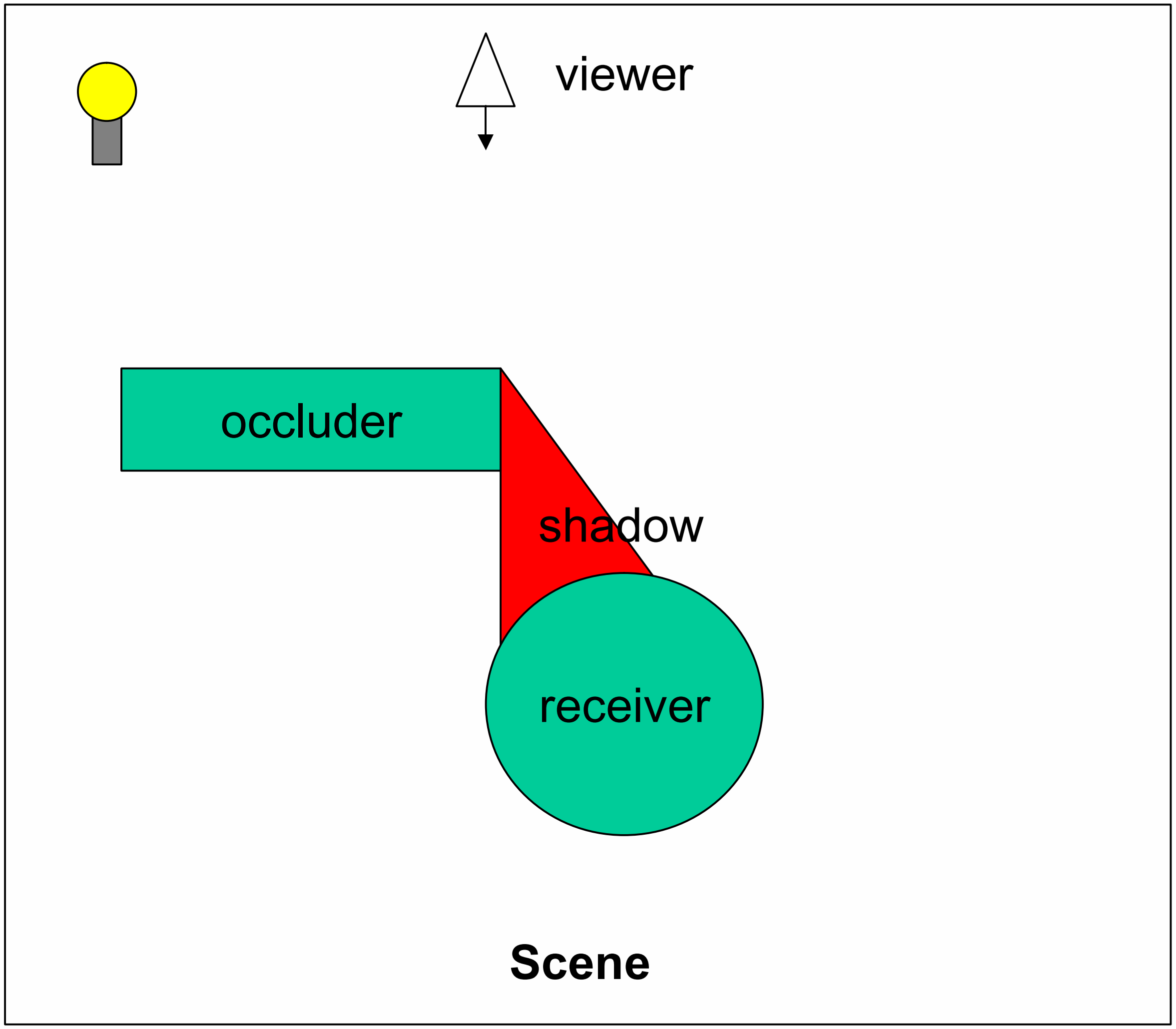
Post-Processing

- G-buffer + accum buffers can be used as input to many post-process effects
  - Glow
  - Auto-Exposure
  - Distortion
  - Edge-smoothing
  - Fog
  - Whatever else!
  - HDR
- See HDR talk

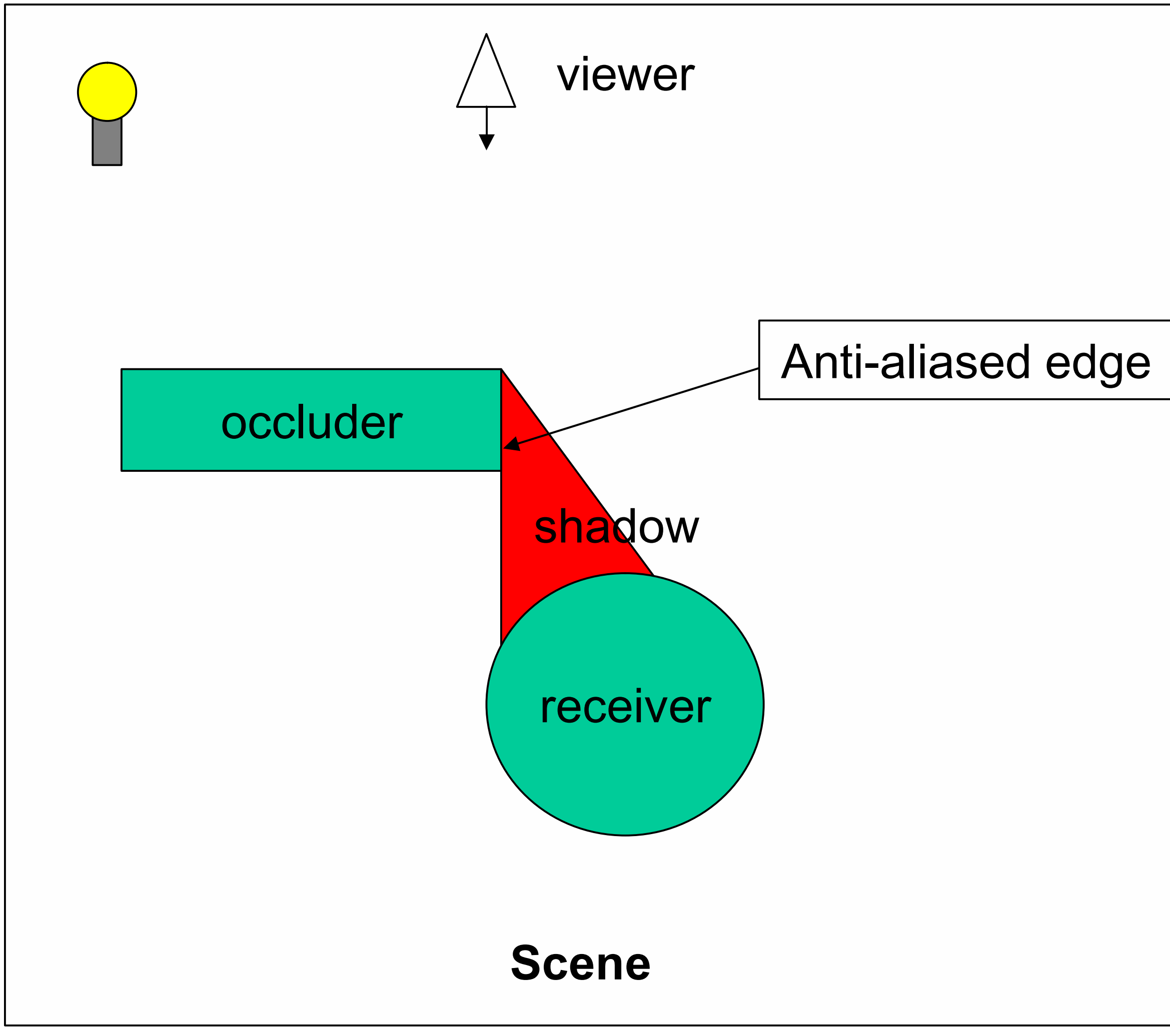
Anti-Aliasing with Deferred Shading

- Deferred shading is incompatible with MSAA
- API doesn't allow antialiased MRTs
  - But this is a small problem…
- AA resolve has to happen after accumulation!
  - Resolve = process of combining multiple samples
- G-Buffer cannot be resolved
  - What happens to an FP16 position when resolved?

Shadow Edge, Correct AA Resolve



Shadow Edge, Correct AA Resolve

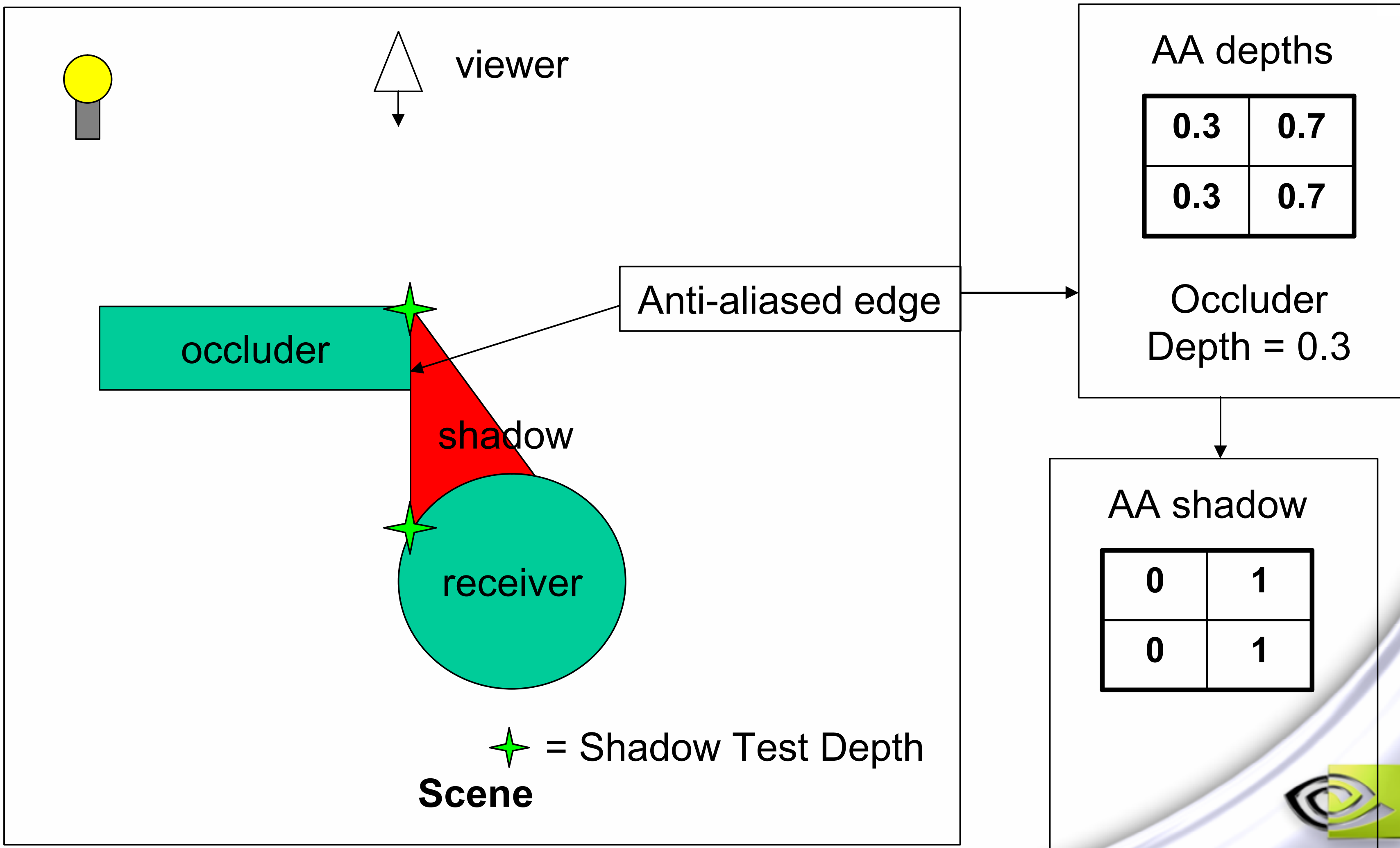


AA depths

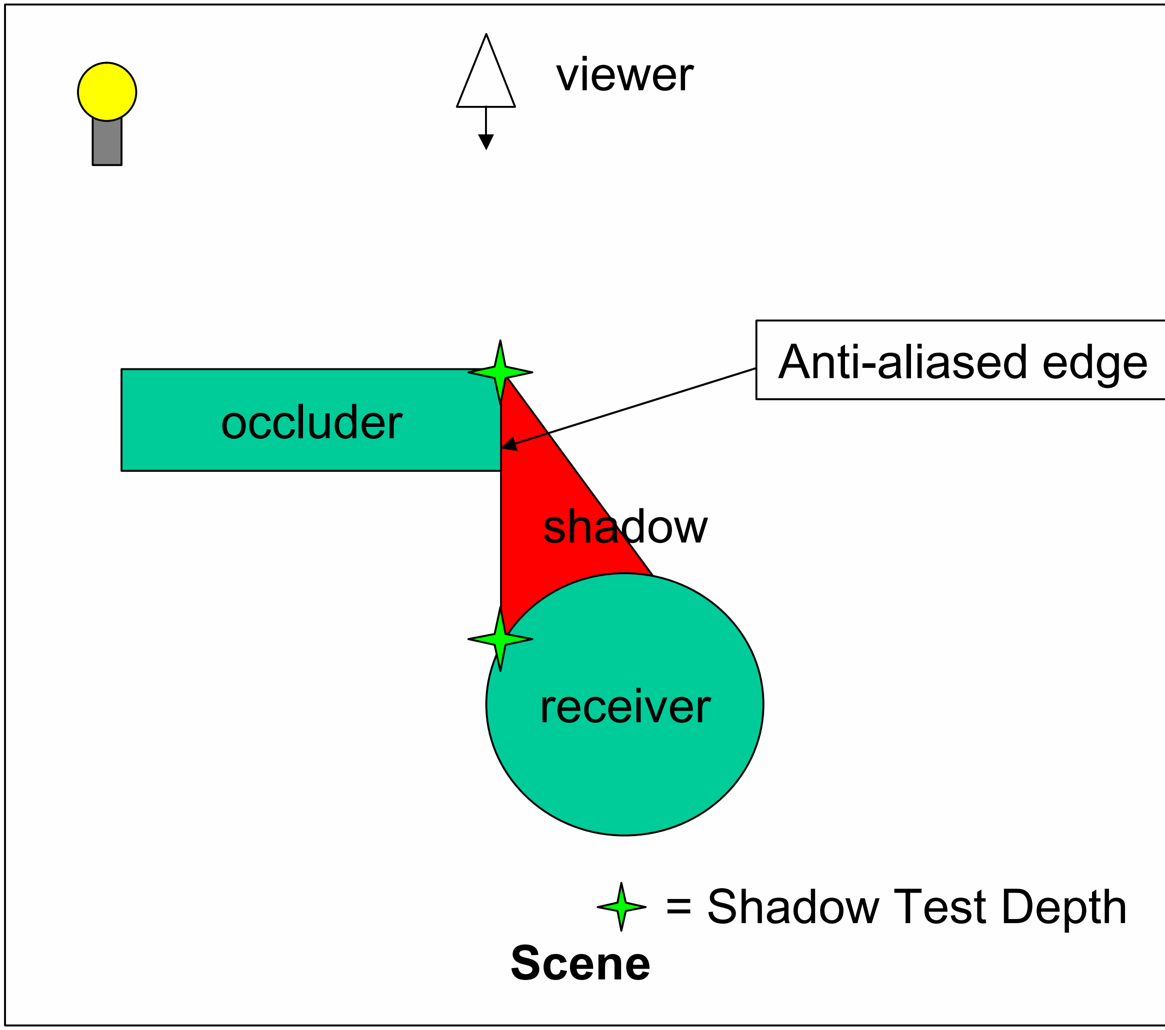
0.3	**0.7**
0.3	**0.7**

Occluder
Depth = 0.3

Shadow Edge, Correct AA Resolve



Shadow Edge, Correct AA Resolve



AA depths

0.3	**0.7**
0.3	**0.7**

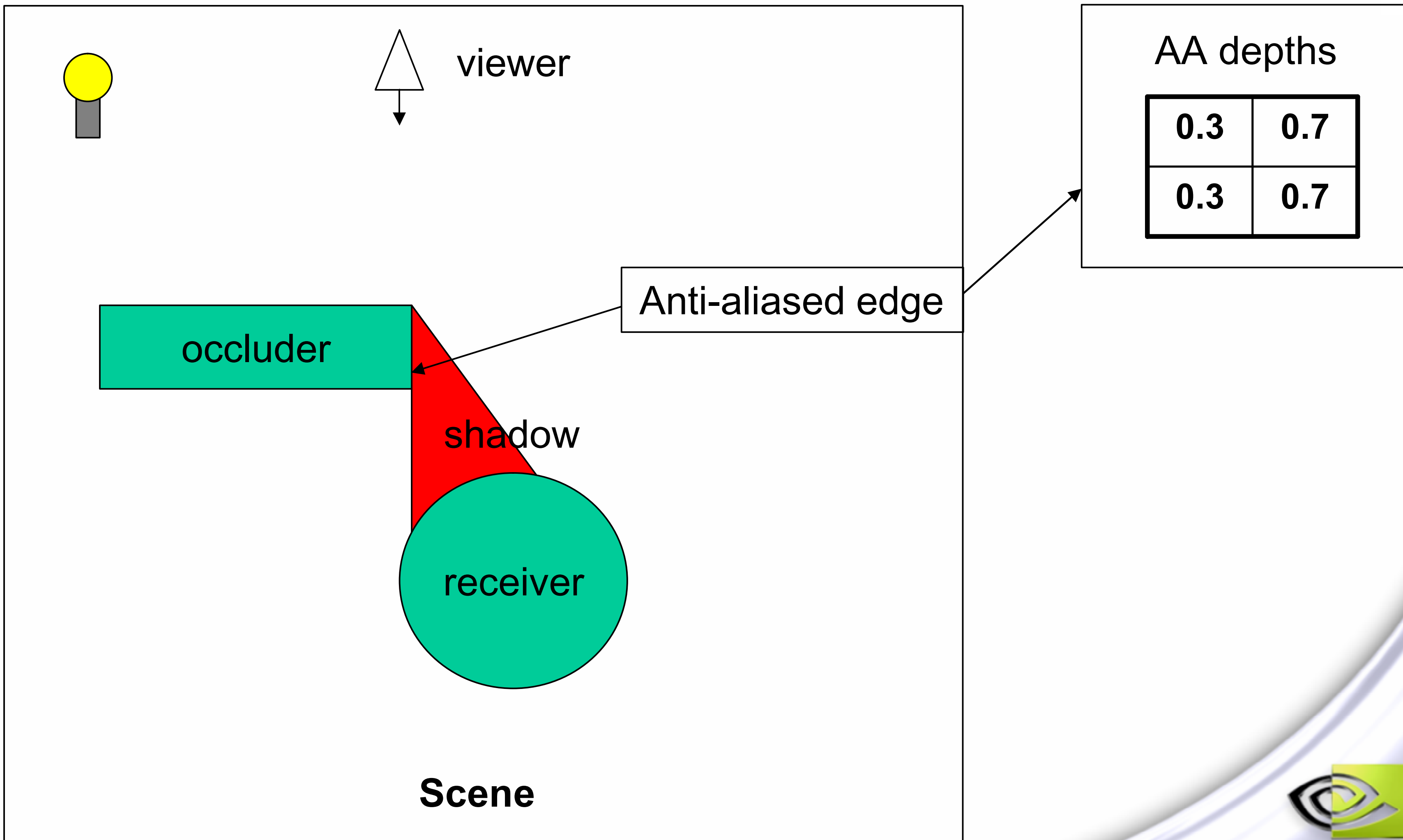
Occluder
Depth = 0.3

AA shadow

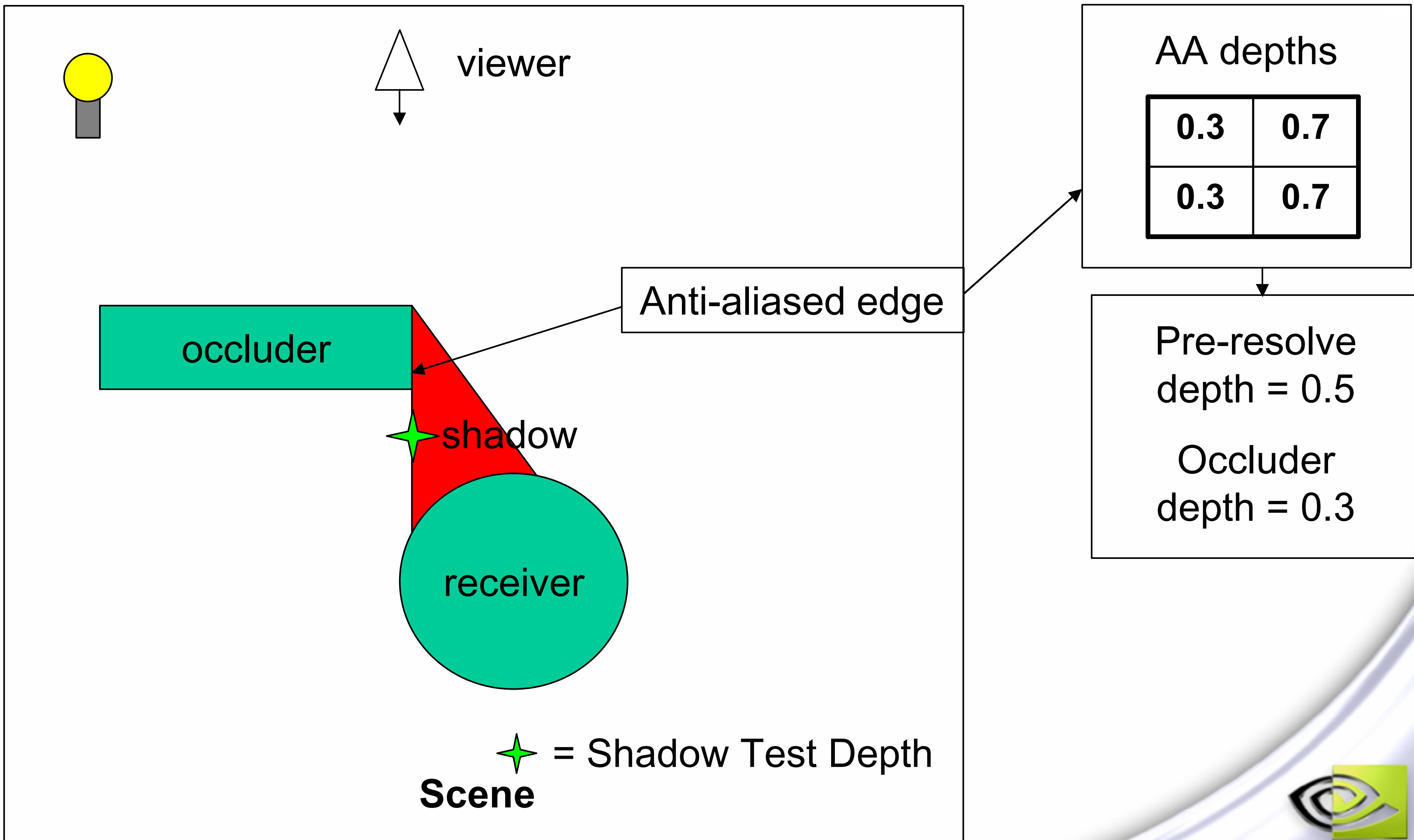
0	**1**
0	**1**

shadow = **0.5**

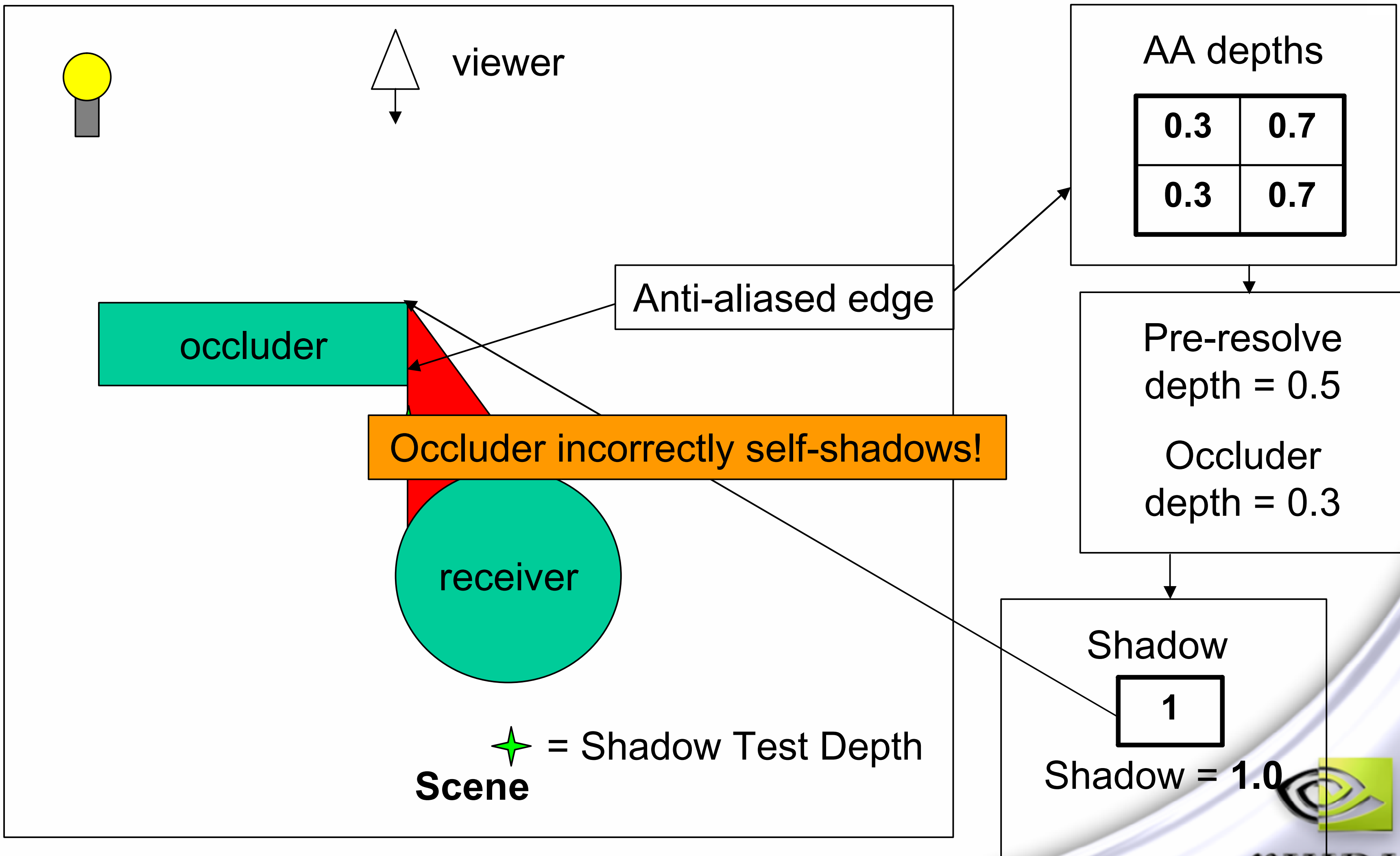
Shadow Edge, G-Buffer Resolve



Shadow Edge, G-Buffer Resolve



Shadow Edge, G-Buffer Resolve



Other AA options?

- Supersampling lighting is a costly option
  - Lighting is typically the bottleneck, pixel shader bound
  - 4x supersampled lighting would be a big perf. Hit
- “Intelligent Blur” : Only filter object edges
  - Based on depths and normals of neighboring pixels
  - Set “barrier” high, to avoid interior blurring
  - Full-screen shader, but cheaper than SSAA

Should I use Deferred Shading?

- This is an ESSENTIAL question
- Deferred shading is not always a win
  - One major title has already scrapped it!
  - Another came close
- Many tradeoffs
  - AA is problematic
  - Some scenes work well, others very poorly
- The benefit will depend on your application
  - Game design
  - Level design

When is Deferred Shading A Win?

- Not when you have many directional lights
  - Shading complexity will be O(R*L), R = screen res.
  - Outdoor daytime scenes probably not a good case
- Better when you have lots of local lights
  - Ideal case is non-overlapping lights
  - Shading complexity O(R)
  - Nighttime scenes with many dynamic lights!

- In any case, make sure G-Buffer pass is cheap

Gosh, what about z-cull & SM3.0?

- Isn't the goal of z-cull to achieve deferred shading?
  - Do an initial front-to-back-sorted z-only pass.
  - Then you will shade only visible surfaces anyway!
- Shader Model 3.0 allows "uber shaders"
  - Iterate over multiple lights of different types in "traditional" (non-deferred) shading
- Combine these, and performance could be as good (better?) than deferred shading!
  - More tests needed

We don’t have all the answers

- We can’t tell you to use it or not
  - Experimentation and analysis is important
  - Depends on your application
  - Need to have a fallback anyway

Sorry to end it this way, but…

MORE RESEARCH IS NEEDED!

PLEASE SHARE YOUR FINDINGS!

(you can bet we'll share ours)

Questions?

- http://developer.nvidia.com
- mharris@nvidia.com

GeForce 6800 Guidance (1 of 6)

- Allocate render targets FIRST
  - Deferred Shading uses many RTs
  - Allocating them first ensures they are in fastest RAM

- Keep MRT usage to 3 or fewer render targets
  - Performance cliff at 4 on GeForce 6800
  - Each additional RT adds shader overhead
  - Don’t render to all RTs if surface doesn’t need them
    - e.g. Sky Dome doesn’t need normals or position

GeForce 6800 Guidance (2 of 6)

- Use aniso filtering during G-buffer pass
  - Will help image quality on parts of image that don’t benefit from “edge smoothing AA”
  - Only on textures that need it!
- Take advantage of early Z- and Stencil culling
  - Don’t switch z-test direction mid-frame
  - Avoid frequent stencil reference / op changes

GeForce 6800 Guidance (3 of 6)

- **Use hardware shadow mapping (“UltraShadow”)**
  - **Use D16 or D24X8 format for shadow maps**
  - **Bind 8-bit color RT, disable color writes on updates**
  - **Use tex2Dproj to get hardware shadow comparison**
  - **Enable bilinear filtering to get 4-sample PCF**

GeForce 6800 Guidance (4 of 6)

- **Use fp16 filtering and blending**
  - **Fp16 textures are fully orthogonal!**
  - **No need to “ping-pong” to accumulate light sources**

- **Use the lowest precision possible**
  - **Lower-precision textures improve cache coherence, reduce bandwidth**
  - **Use half data type in shaders**

GeForce 6800 Guidance (5 of 6)

- Use write masks to tell optimizer sizes of operands
  - Can schedule multiple instructions per cycle
    - Two simultaneous 2-component ops, or
    - One 3-component op + 1 scalar op
- Without write masks, compiler must be conservative

GeForce 6800 Guidance (6 of 6)

- Use fp16 normalize()
  - Compiles to single-cycle nrmh instruction
  - Only applies to half3, so:

```
half3 n = normalize(tex2D(normalmap, coords).xyz);  // fast
half4 n = normalize(tex2D(normalmap, coords));      // slow
float3 n = normalize(tex2D(normalmap, coords).xyz); // slow
```

Example Attribute Layout

- Normal: x,y,z
- Position: x, y, z
- Diffuse Reflectance: RGB
- Specular Reflectance ("Gloss Map", single channel)
- Emissive (single channel)
- One free channel
  - Ideas on this later
  - Your application will dictate