Special Effects with DirectX 9

Alex Vlachos
AVlachos@ati.com

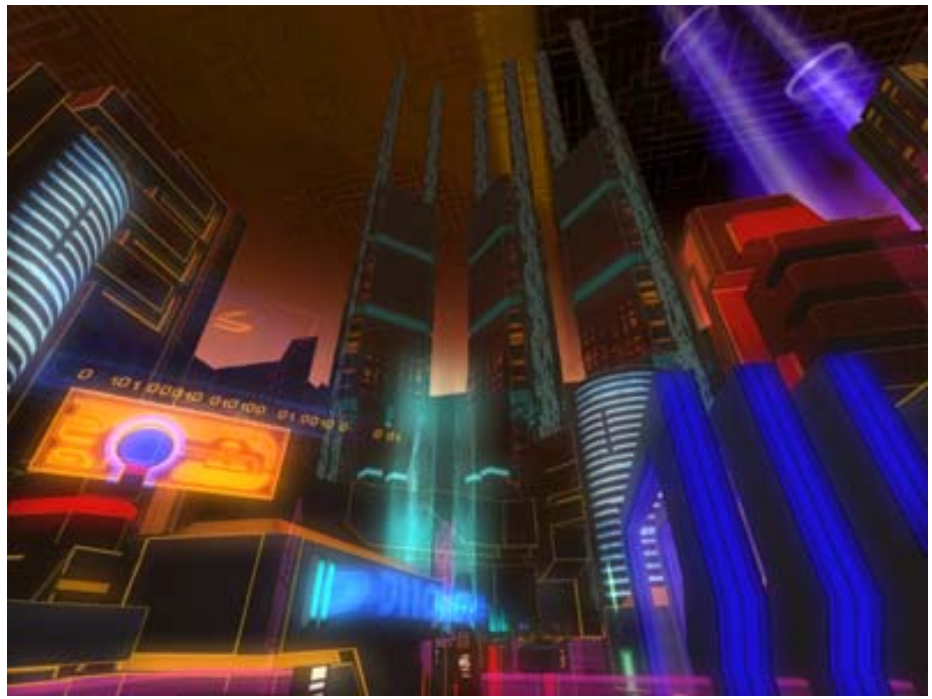
Greg James
GJames@nvidia.com

Outline

- Glow effect
  - Developed for Disney/Monolith's "Tron 2.0"
- Volume fog from polygon objects
  - Used in Bandai/Dimps "UniversalCentury.net Gundam Online"
- Shadows in the Animusic demo
  - Composite and Static shadows
  - Shadow color determination
- Real-Time Fur
  - Coloring
  - Thinning
  - Length culling

“Tron 2.0” Glow Effect

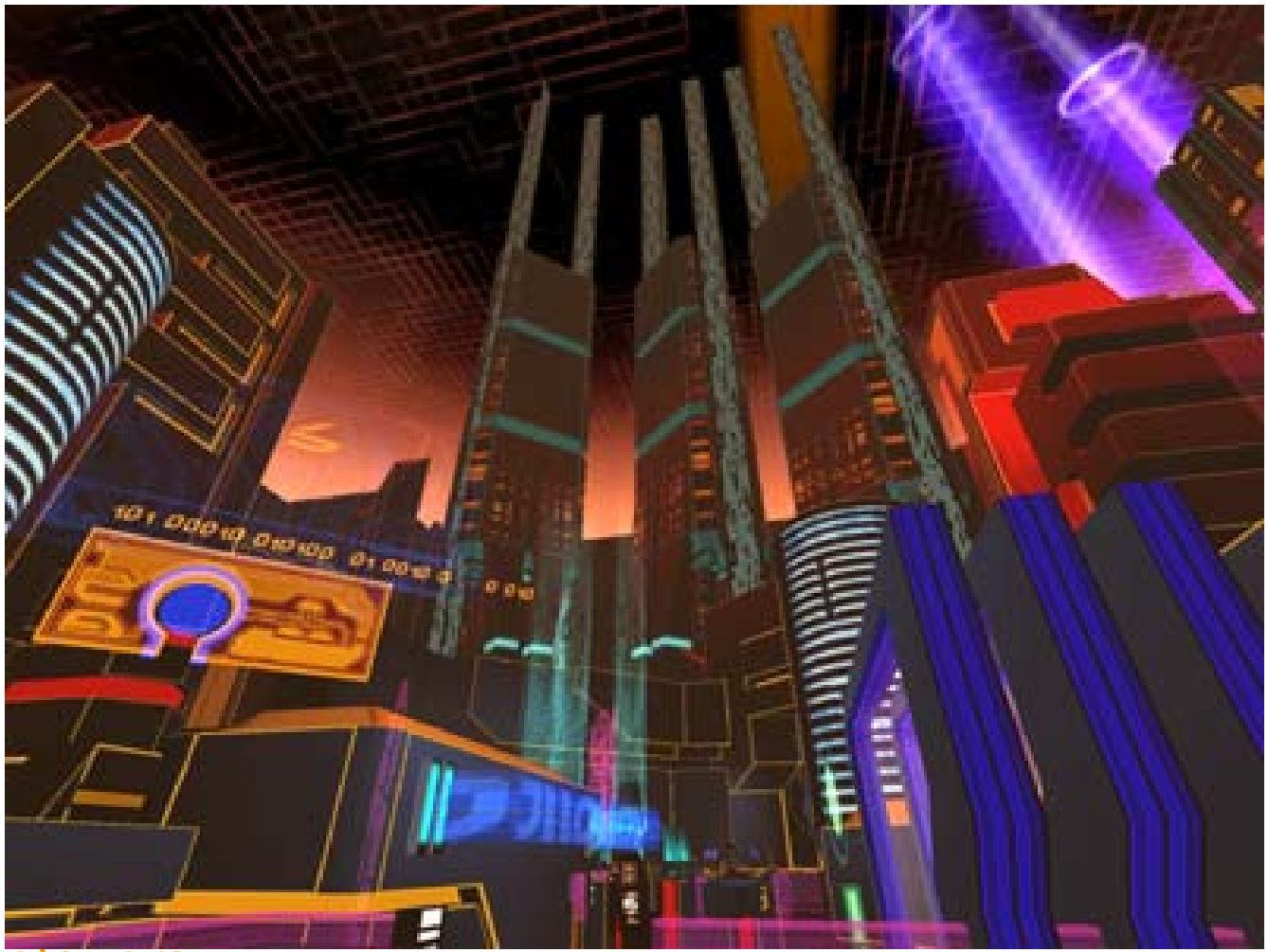
- Large glows for complex scenes
- Fast for in-game use in a FPS
- Efficient HDR effect
- Multi-colored glow
- Easy to control



“Tron2.0” courtesy of Monolith & Disney Interactive

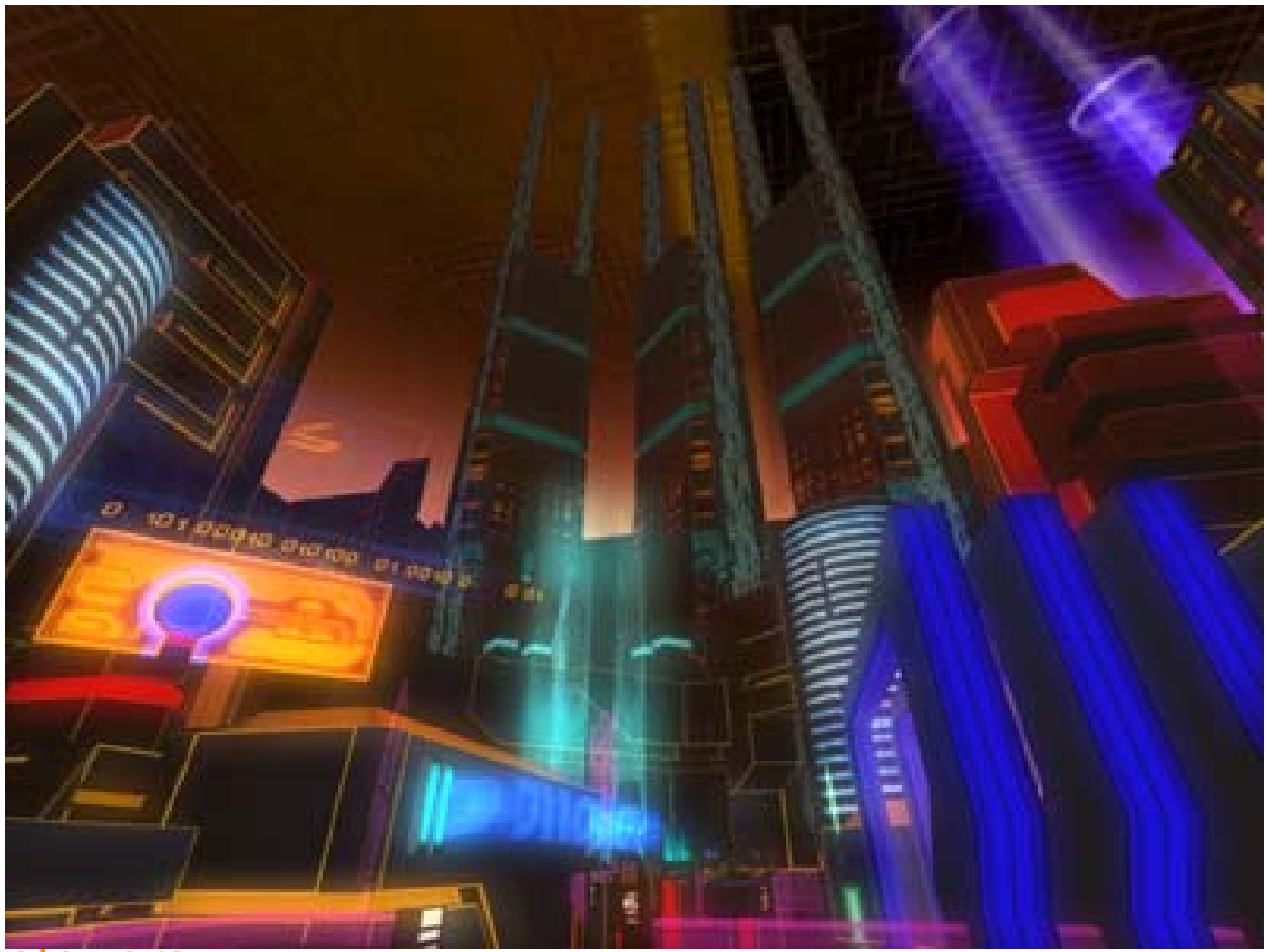
No Glow

"Tron2.0" courtesy of Monolith & Disney Interactive



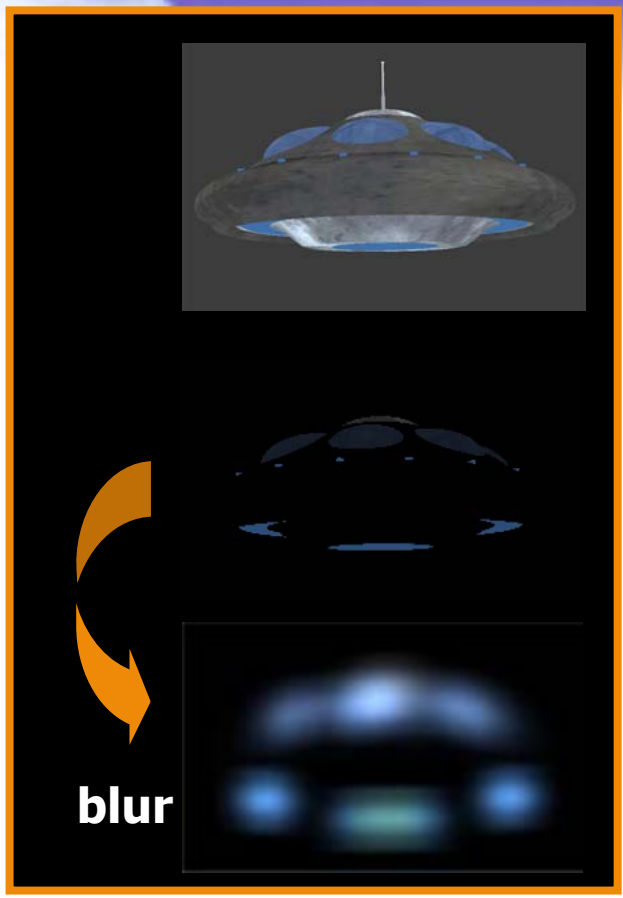
Glow

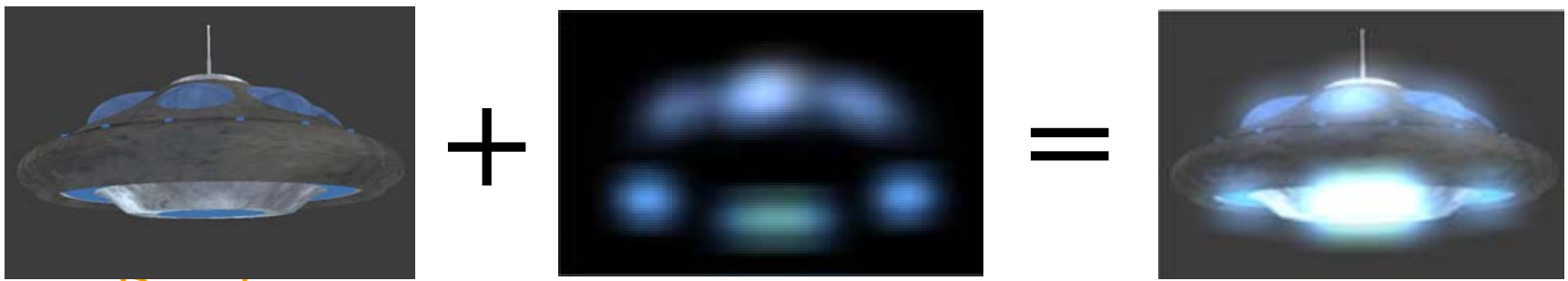
"Tron2.0" courtesy of Monolith & Disney Interactive



How It Works

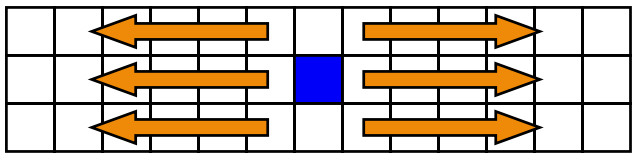
- Start with ordinary model
  - Render to backbuffer
- Render parts that are the sources of glow
  - Render to offscreen texture
- Blur the texture
- Add blur to the scene



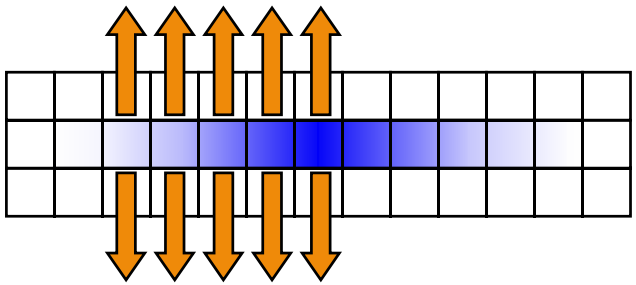


Efficient Blur

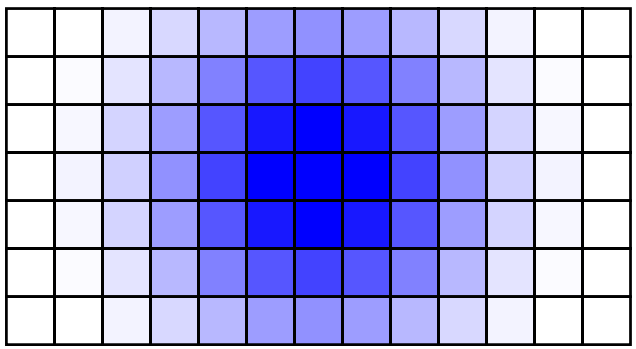
- **Blur, then blur the blur**



Blur the source horizontally



Blur the blur verticaly

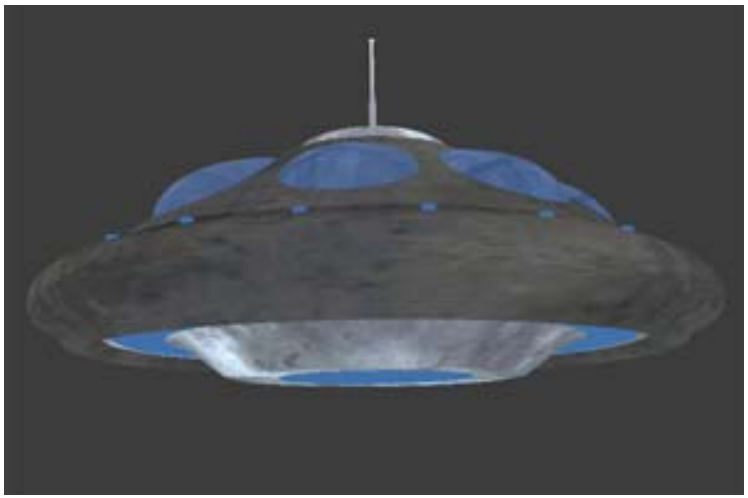


Result

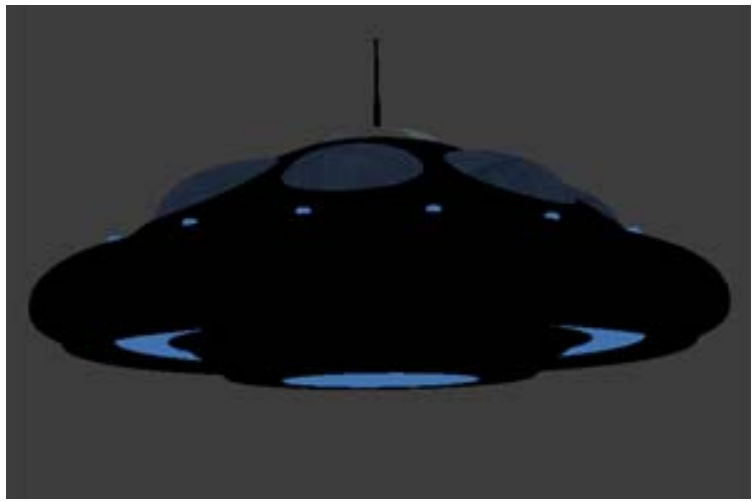
General Approach

- No CPU pixel processing!
  - No texture locks or CPU readbacks
  - Render to GPU video memory textures
- Minimize render target changes
- Fill rate bound
  - Minimize fill cost
  - Low resolution glow processing
  - Magnify glow texture to cover the full screen
- Full scene gets blurred at once
  - Could break it up for finer control

Specify Glow Sources



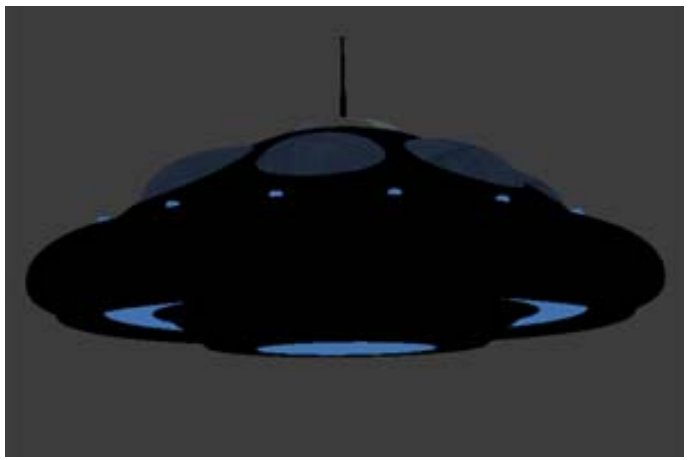
Model with diffuse texture, t0.rgba



t0.a * t0.rgb = glow source

- Start with ordinary model
- Designate areas as 'glow sources'
  - texture Alpha * texture RGB = glow source color
  - or create separate glow geometry

Render Glow Sources to Texture



Texture render target

- Texture render target can be lower resolution than final display
  - Glows are low frequency, smooth
  - Can be rendered at low resolution
  - The lower the resolution, the more aliased the sources
    - You can miss glow sources
    - Glow may shimmer and flicker

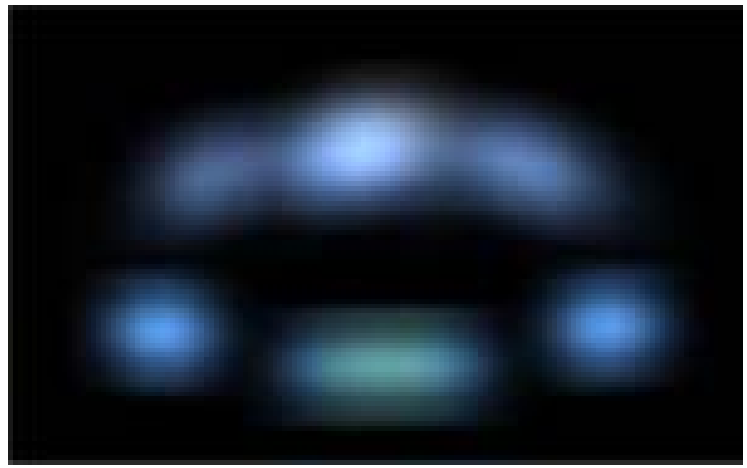
Low Texture Resolution

- Improve performance and size of glows
- Each glow texel can cover 2, 3, 4 etc. screen pixels
  - Example: Blur a 40x40 texel area
  - Becomes a 160x160 screen pixel glow

Assets courtesy of Monolith & Disney Interactive

Blur to Create Glow Texture

Rendered Texture



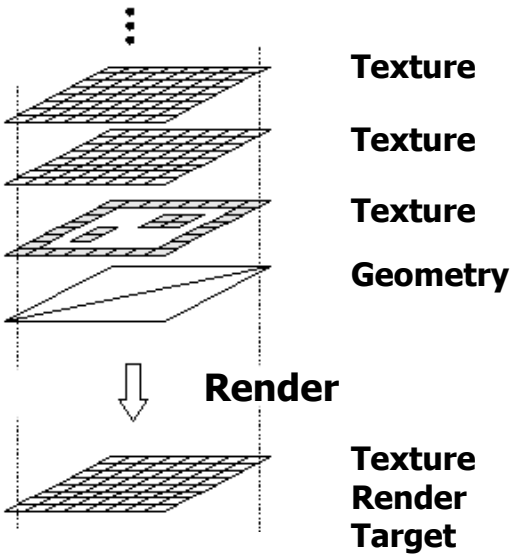
Texture render target

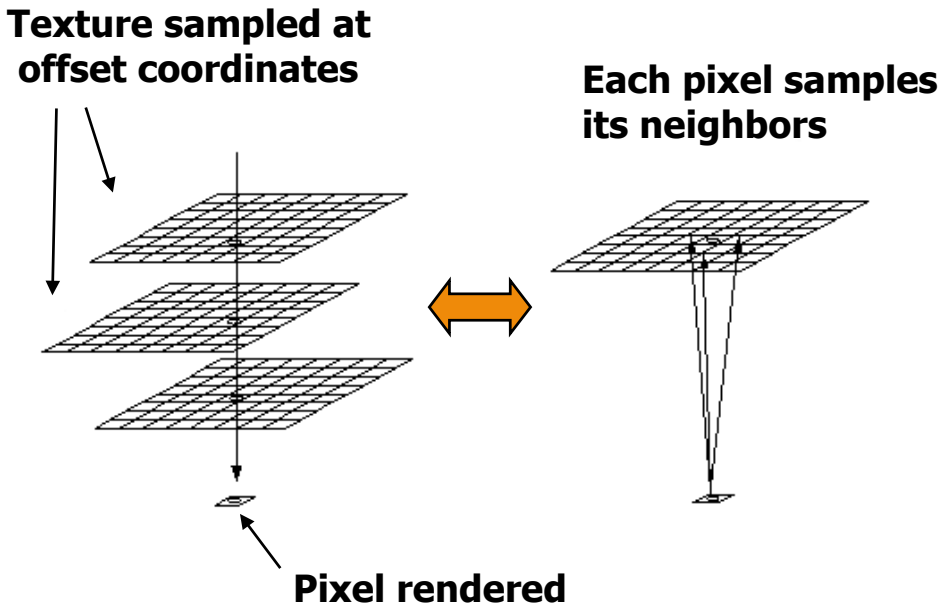
- GPU render-to-texture
- Pixel samples from many neighbors
  - Details: "Game Programming Gems 2" article

    "Operations for HW-Accelerated Procedural Texture Animation"

How To Blur

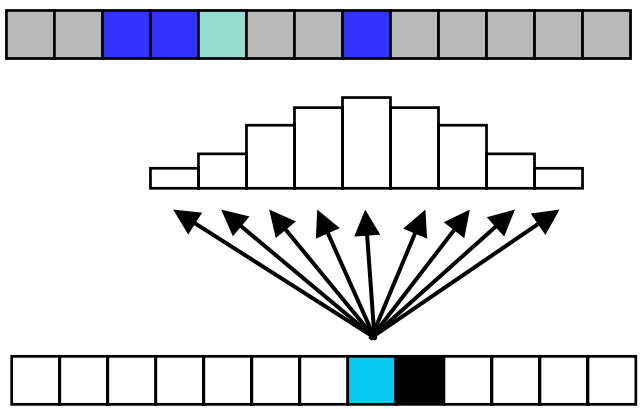
- Neighbor sampling with vertex and pixel shaders
- Simple geometry with several texture coordinates
- Each tex coord samples a pixel's neighbor





How to Blur in One Axis

- **D3D9**
  - **Use 1 bound texture, sampled N times**
  - **Each sample multiplied by blur profile weight**
  - **Single pass**



Glow source texture

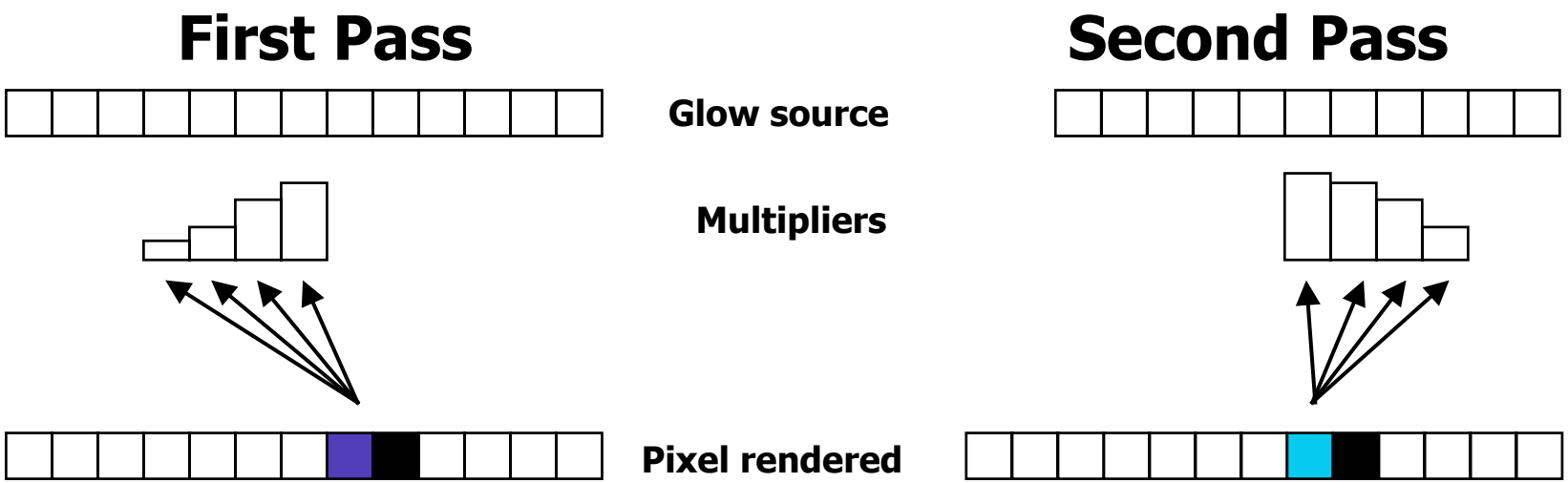
Multiplier (weight)

Neighbor sampling using texture coordinates

Pixel being rendered

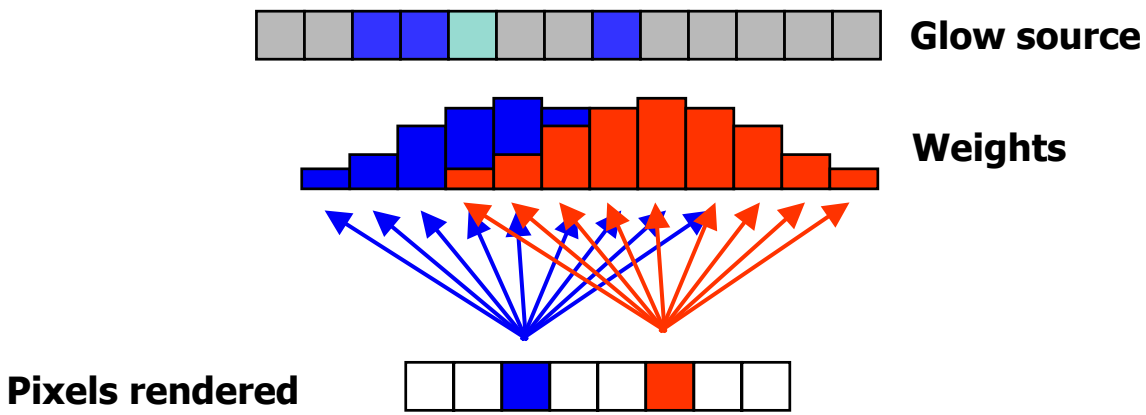
How to Blur With D3D8 HW

- **D3D8**
  - **Multiple additive passes to build up N samples**
  - **Bind source to 4 tex units, each sampled once**
  - **4 samples per pass, point or bilinear sampled**



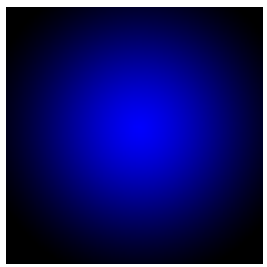
Neighbor Sampling

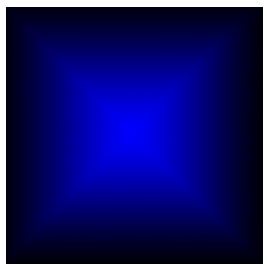
- Each pixel samples the same pattern of neighbors
- One D3D9 pass blurs all pixels horizontally
- One more pass blurs all pixels vertically



Blurring

- You might hear ‘separable Gaussian’
- We can use any blur profiles
  - More than just Gaussian
- Separating into BlurV(BlurH(x)) restricts the 2D blur shapes
  - Good shapes still possible
  - Watch for square features





Add Glow to Scene

- Apply glow using two triangles covering the screen
- Additive blend

Performance Concerns

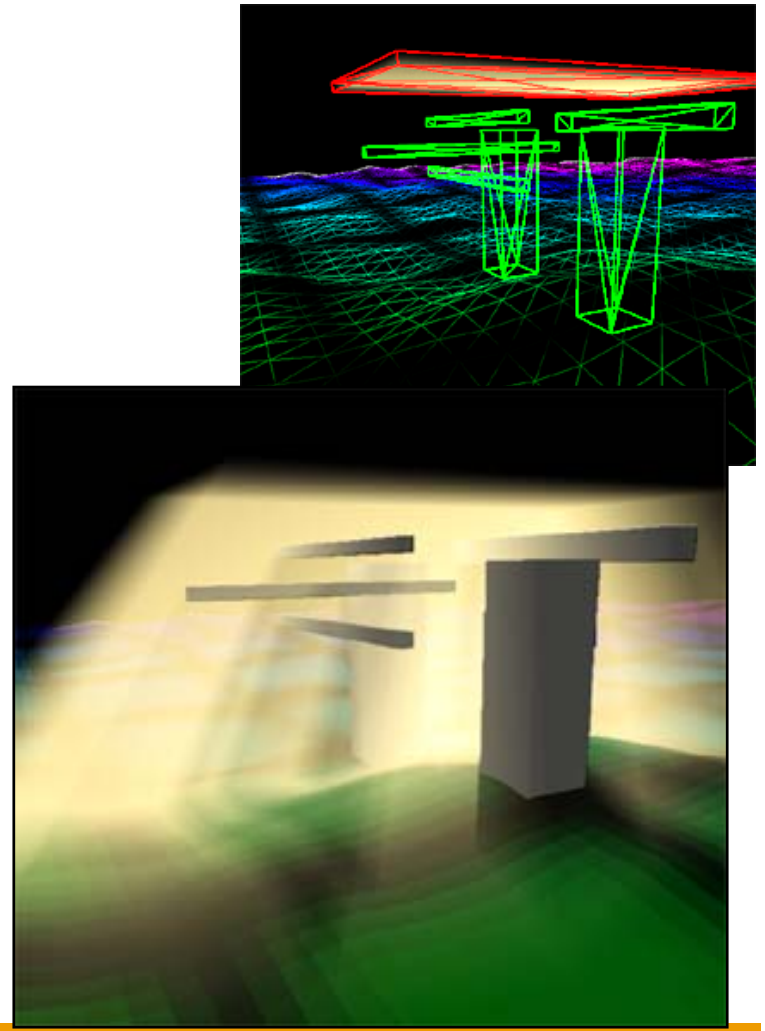
- Limited by:
- Number of DrawPrimitive calls needed to render glow sources
  - Batch rendering of glow sources as much as possible
  - Call Draw..Primitive() as little as possible
- Texture render target resolution
  - Use pow2 textures or non-pow2? 256x256 or 300x200?
  - Test each
- Blur convolution size
  - Perf of NxN separable blur is O(N), not $O(N^2)$ ☺

Many Uses for Glow Technique

- Key to making things look bright
  - Subtle glow has dramatic effect
  - Reflections: water, shiny objects
  - Atmospheric: neon lights, smoke haze
- More than just glow!
  - Blur, depth of field, light scattering
- Remember, it doesn't require HDR assets or floating point textures!
  - Great for D3D8 hardware
  - Greater with D3D9 hardware

Volume Fog from Polygon Hulls

- Polygon hulls rendered as thick volumes
- True volumetric effect
- Very easy to author
- Animate volume objects
- Positive and negative volumes
- Fast, efficient occlusion & intersection
- ps_2_0, ps.1.3 fallbacks



Practical Effect

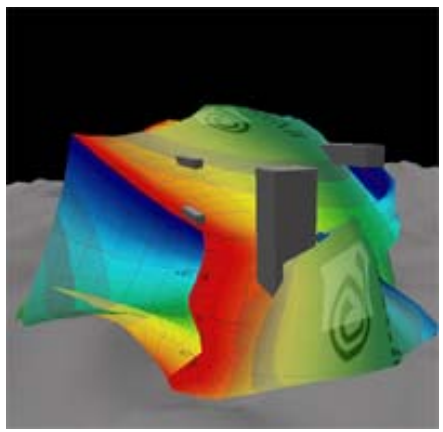
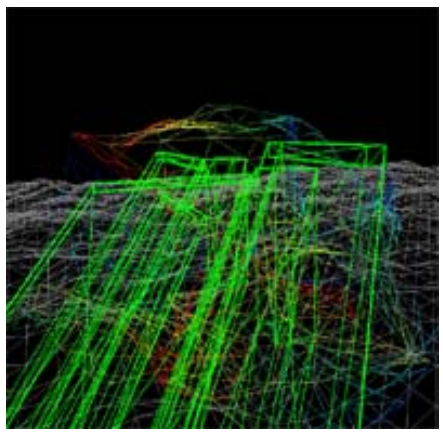
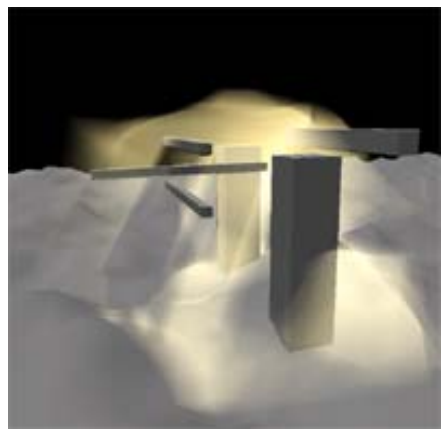
- **Used in Bandai/Dimps "UniversalCentury.net Gundam Online"**
  - **Engine thrust**

Concept art

In-game

Volume Objects

- **Ordinary polygon hulls**
  - **Use existing objects. Closed hulls**
  - **No new per-object vertex or pixel data**
  - **Just a scale value for thickness-to-color and 3 small shared textures**
  - **Can use stencil shadow volume geometry**

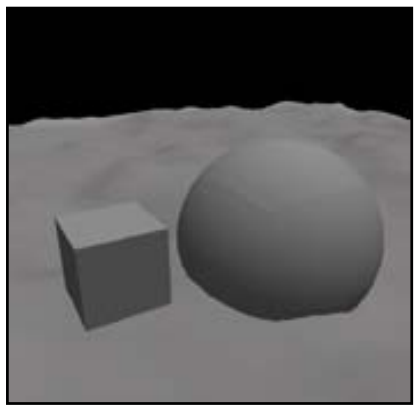
The Technique

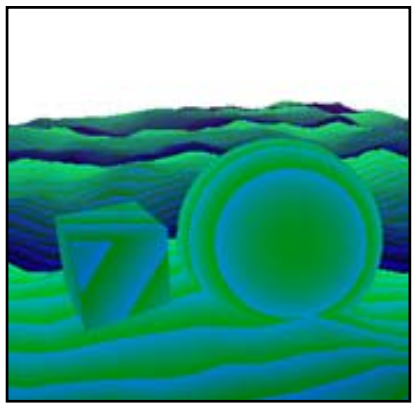
- Inspired by Microsoft's "Volume Fog" DXSDK demo
- Improves the approach
  - Higher precision: 12, 15, 18, 21-bit depth
  - Precision vs. depth complexity tradeoff
  - High precision decode & depth compare
  - Dithering
  - No banding, even with deep view frustum
  - Simple, complete intersection handling for any shapes

The Technique

- Render to offscreen textures
- Instead of rendering object "color," render the object depth at each pixel
  - Encode depth as RGB color

Objects





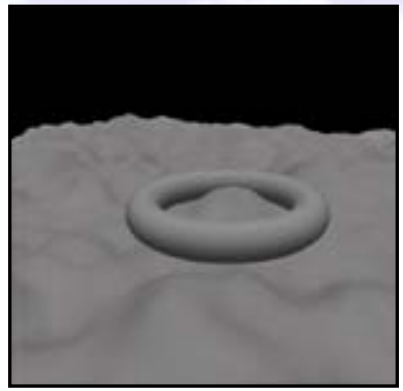
RGB-encoded depth

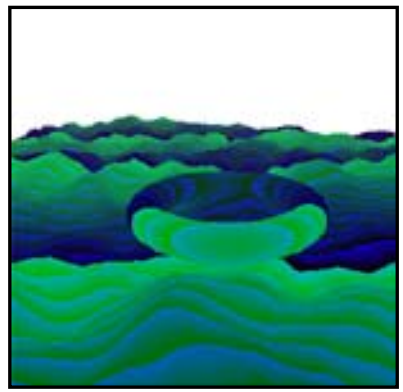
- Depths used to calculate thickness through objects at each pixel

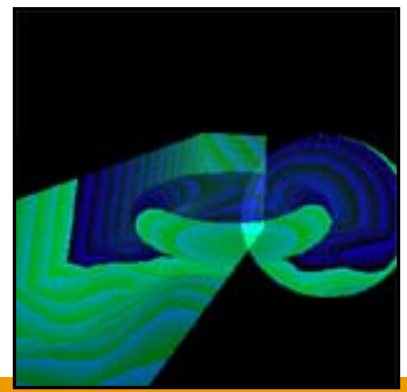
Before all the Details…

Here's how simple it is!

1. Render solid objects to backbuffer
   - Ordinary rendering
2. Render depth of solid objects that might intersect the fog volumes
   - To ARGB8 texture, "S"
   - RGB-encoded depth. High precision!
3. Render fog volume backfaces
   - To ARGB8 texture, "B"
   - Additive blend to sum depths
   - Sample texture "S" for intersection

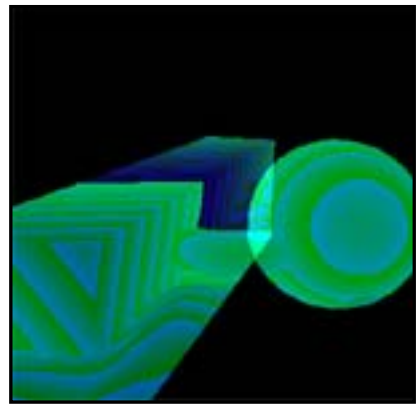


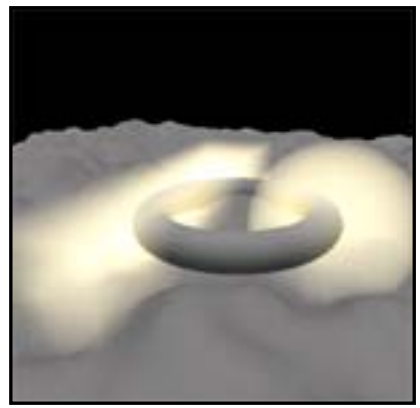




Simplicity…

4. Render fog volume front faces
   - To ARGB8 texture, “F”
   - Additive blend to sum depths
   - Sample texture “S” for intersections
5. Render quad over backbuffer
   - Samples “B” and “F”
   - Computes thickness at each pixel
   - Samples color ramp
   - Converts thickness to color
   - Blends color to the scene
   - 7 instruction ps_2_0 shader



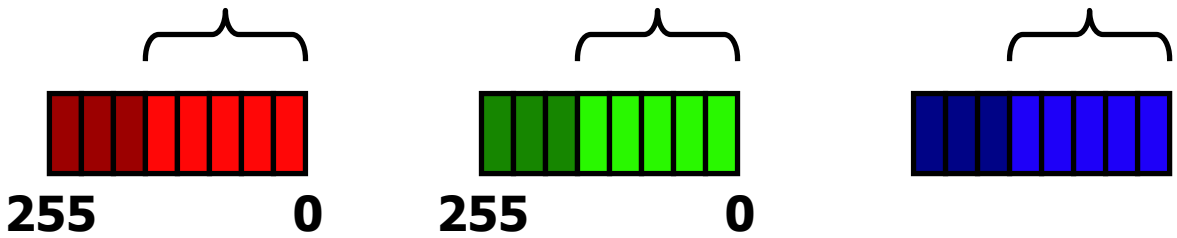


Floating Point Image Surfaces?

- Why not use those?
- Need additive blending
  - No existing HW supports float additive blending to the render target
  - Too many passes without it
- ARGB8 surfaces can do the job
  - Good for all D3D8 pixel shading hardware
  - Millions can run the effect today

RGB-Encoding of Depth

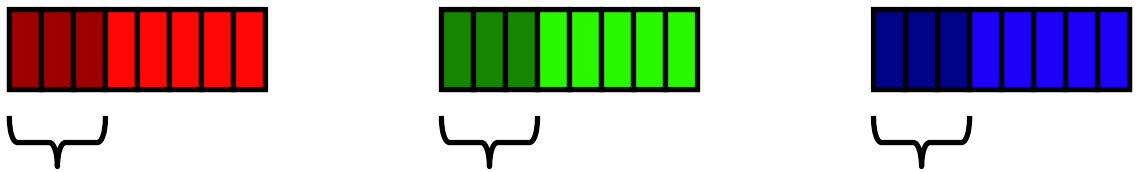
- Use "L" low bits of each color channel
  - ie. 5 low bits from each R, G, and B color
  - Gives 3*L bits of precision (15-bit precision)



- (8 – L) high bits "H" for acumulation
  - $2^{(8-L)}$ depth values can be added before overflow
  - ie. L=5 lets you add 8 values safely

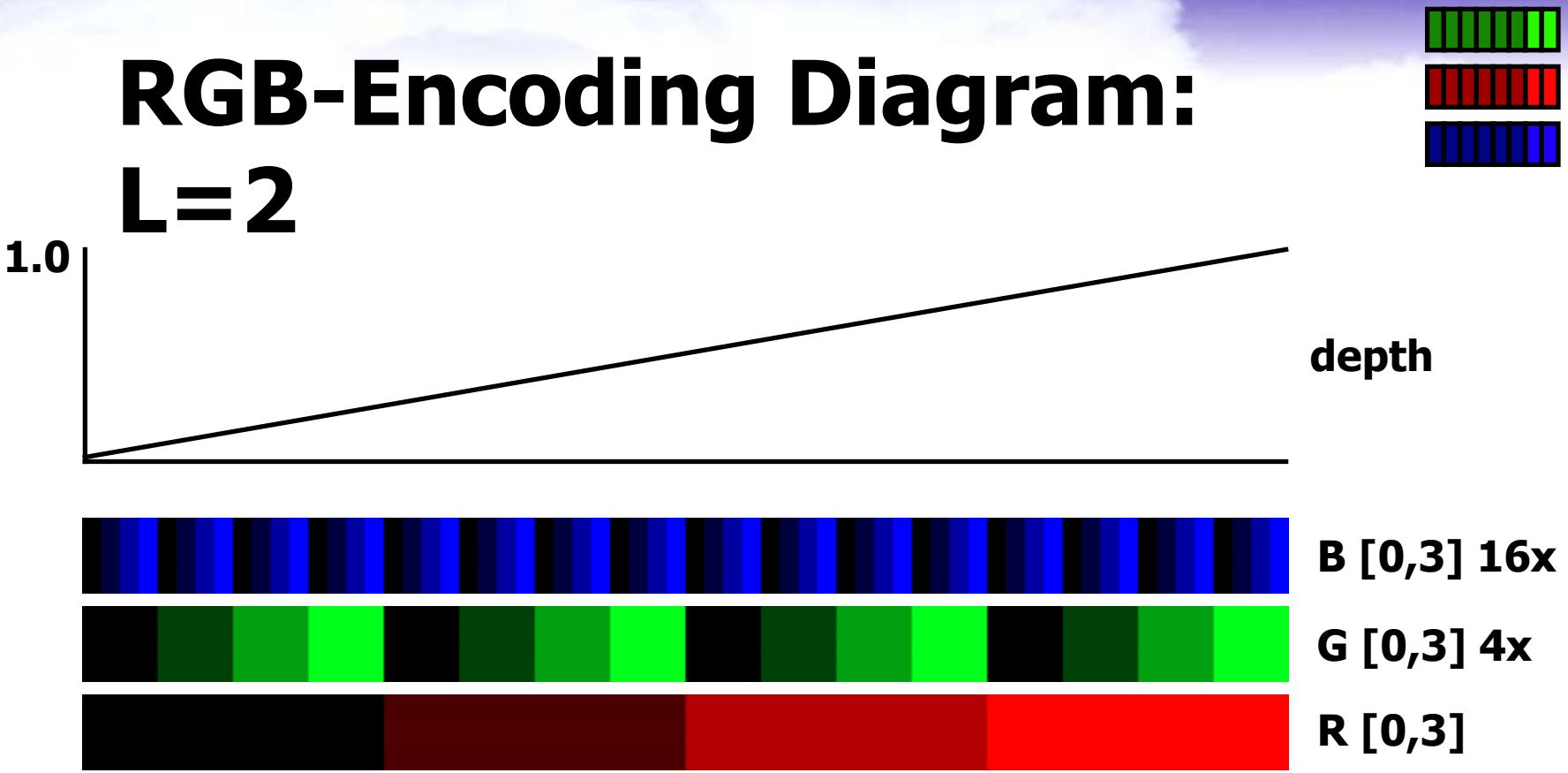
RGB-Encoding of Depth

- Use "L" low bits of each color channel
  - ie. 5 low bits from each R, G, and B color
  - Gives 3*L bits of precision (15-bit precision)



- (8 – L) high bits "H" for summing values
  - $2^{(8-L)}$ values can be added before saturation
  - ie. L=5 lets you add 8 values correctly

RGB-Encoding Diagram: L=2



- One 6-bit depth uses only [0,3] of [0,255]
- Values [4,255] used when adding depths

RGB-Encoding

- **Vertex shader computes depth from [0,1]**

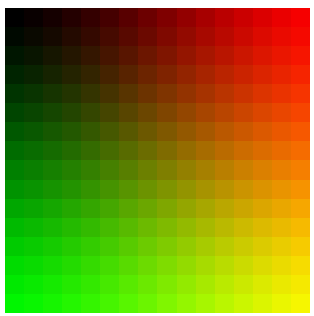
```
DP4 r1.x, V_POSITION, c[CV_WORLDVIEWPROJ_0]
DP4 r1.y, V_POSITION, c[CV_WORLDVIEWPROJ_1]
DP4 r1.z, V_POSITION, c[CV_WORLDVIEWPROJ_2]
DP4 r1.w, V_POSITION, c[CV_WORLDVIEWPROJ_3]
```

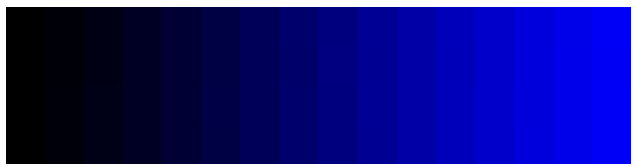
- **Vertes shader turns depth into tex coords**
  - **TexCoord.r = D * 1.0**
  - **TexCoord.g = D * 2^{L} ie. G = D * 16**
  - **TexCoord.b = D * 2^{2L} ie. B = D * 256**

```
MUL  r0.xyz, r1.z, c[CV_DEPTH_TO_TEX_SCALE].xyz
```

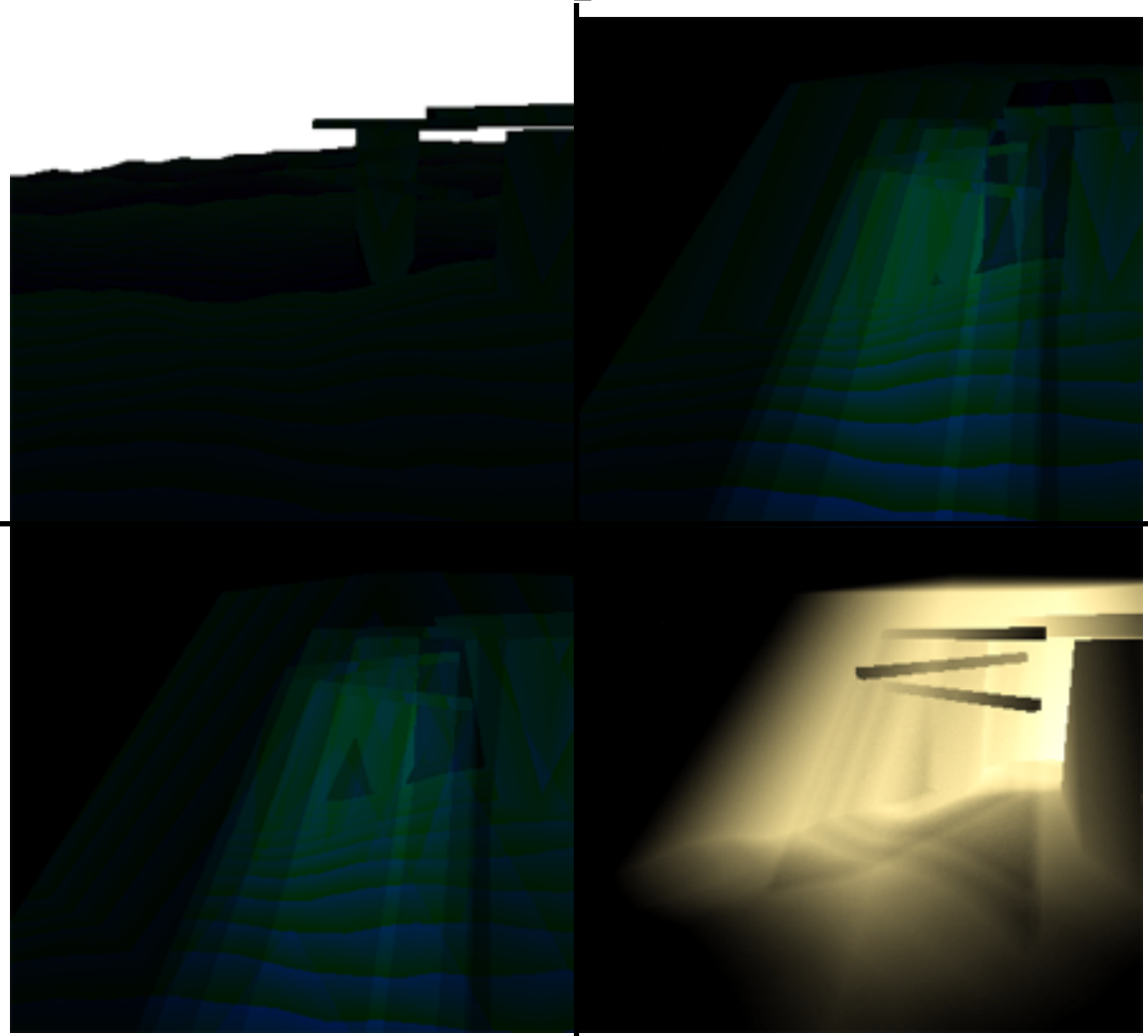
RGB-Encoding

- Texture coordinates read from small R, G, and B ramp textures
  - resolution 2^L in the addressed axis
  - point sampled with wrapping
  - color ramp from $[0, 2^L - 1]$
- Example: L = 4, means 16 values per axis





RGB-Encoded Depths



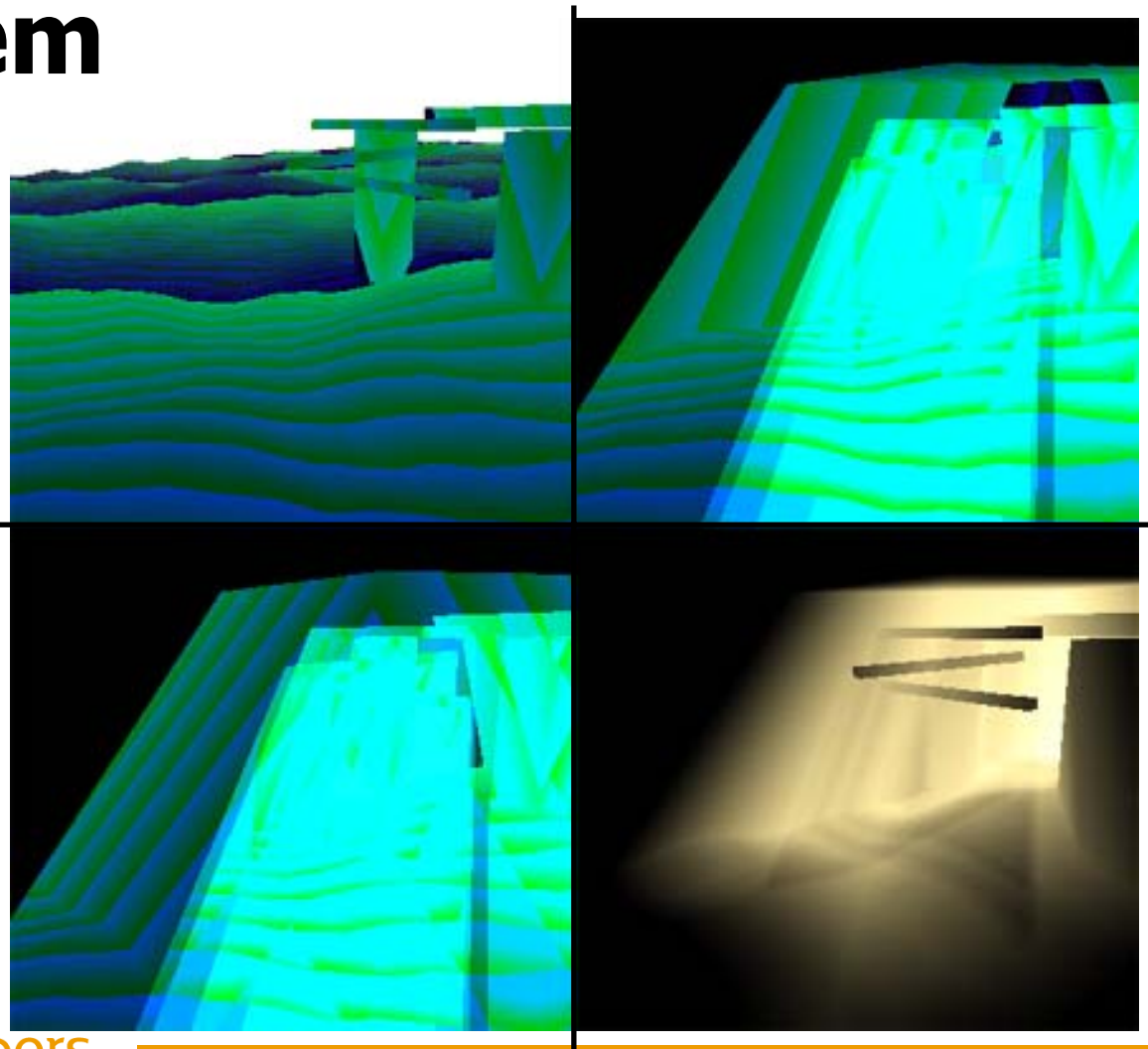
Solid Object Depth

Backfaces Sum

Frontfaces Sum

Rendered Image

Overbright So You Can See Them



Solid Object Depth

Backfaces Sum

Frontfaces Sum

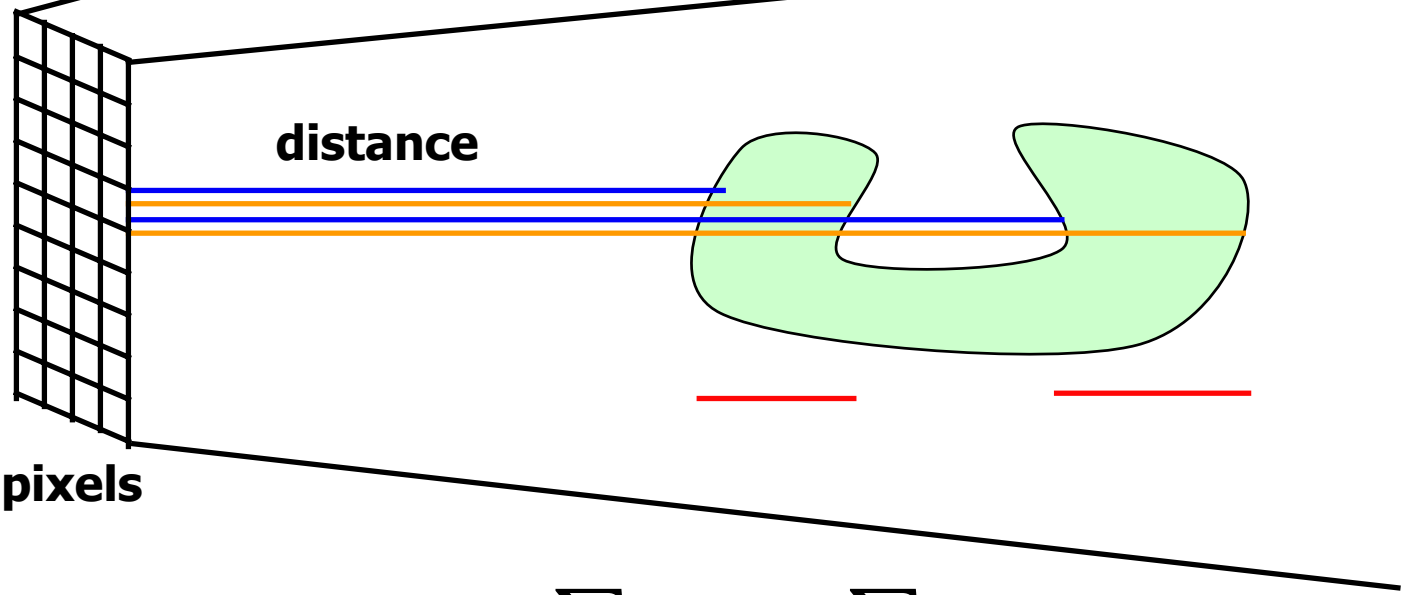
Rendered Image

Precision vs. Number of Surfaces

L low bits	Depth Precision	# of adds
3	9-bit	32
4	12-bit	16
5	15-bit	8

- If visible fog volume depth complexity is higher than the "# of adds" limit:
  - Add a pass to carry the bits
  - Or start rendering to another surface
  - Most likely, this is never needed
- This is using RGB. Could use RGBA

Rendering Thickness Per-Pixel

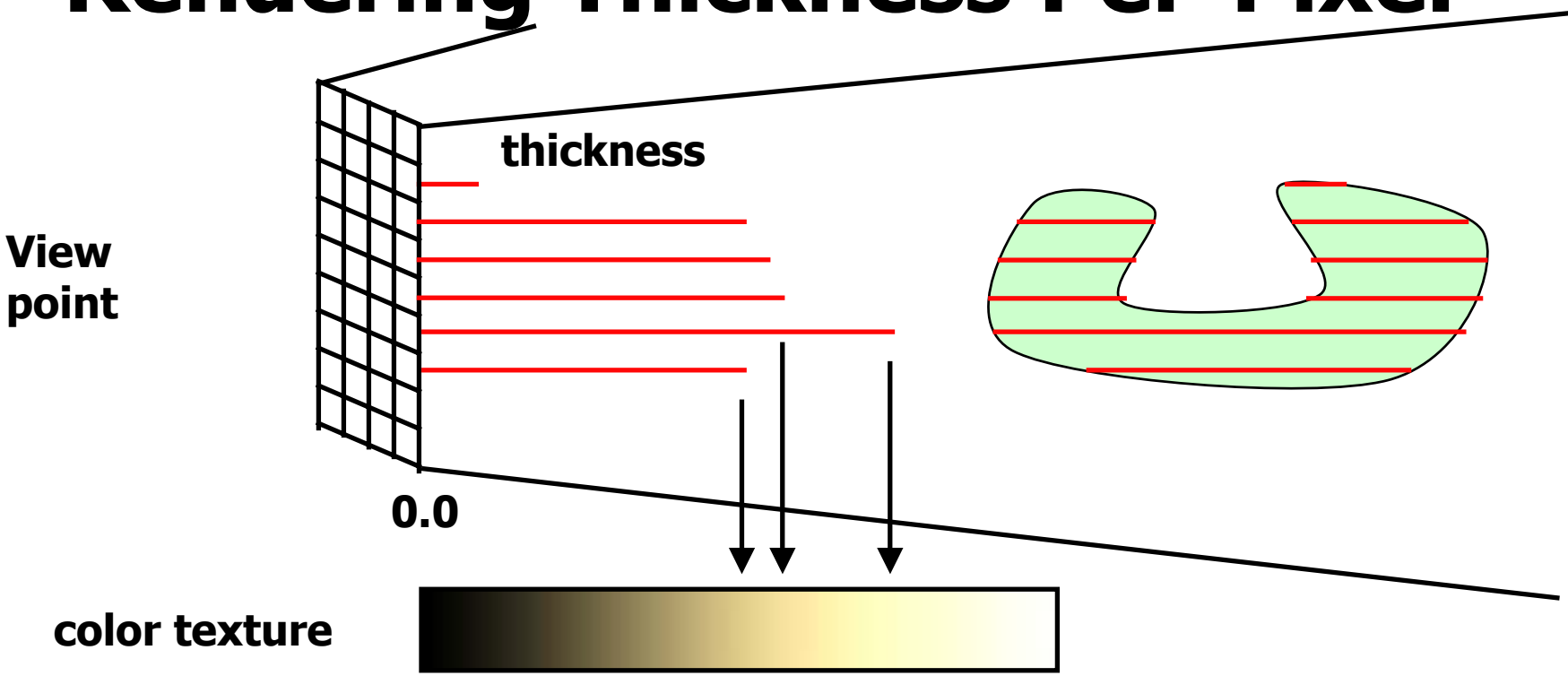


FRONT
BACK
THICKNESS

$$Thickness = \sum Back - \sum Front$$

- Sum the depths of all back faces
- Sum the depths of all front faces
- Difference of the sums is the total thickness

Rendering Thickness Per-Pixel



- Thickness * scale ➔ TexCoord.x
- Artistic or math color ramp
- Very easy to control the look

Decoding RGB-Encoded Values

- **Just one dot-product !**

  Decoded value =
  (D.r, D.g, D.b) DOT (1.0, 2^{-L}, 2^{-2L})

- **Properly handles carried, uncarried, and negative components**
- **Must be done at floating point precision**
  - **ps.1.3 texture address ops**
  - **ps_2_0 shader ops**

Handling Solid Objects Intersecting the Fog

- No additional passes required
- Step 2. texture "S" rendered to have nearest solid object depth
- When rendering fog volume depths:
  - No Z-buffer test. Pixels always written
- Pixel shader:
  - Compute RGB-encoded distance, "D" to pixel
  - Read "S" depth at pixel location
  - If "D" is GREATER than "S" then output "S"
    ELSE output "D"

D3D9 Depth Encode, Compare, and Decision Pixel Shader

```
texld     r0, t0, s0                    // red+green part of depth encoding
texld     r1, t1, s1                    // blue part of depth encoding
ADD       r0, r0, r1                    // RGB-encoded depth of triangle's pixel
texldp    r1, t3, s3                    // RGB-encoded depth from texture at s2

// Compare depth of triangle's pixel (r0) to depth from texture (r1)
// and choose the lesser value to output.

ADD       r2, r0, -r1                   // RGB-encoded difference

                                        // Decode to positive or negative value
DP3       r2, r2,   CPN_RGB_TEXADDR_WEIGHTS

                                        // always choose the lesser value
CMP       r3, r2.xxxx, r1, r0 // r1 >= 0 ? : r1 : r0
MOV       oC0, r3
```

D3D8 Depth Encode, Compare, and Decision Pixel Shader

- **Numbers must saturate to [-1,1] range**

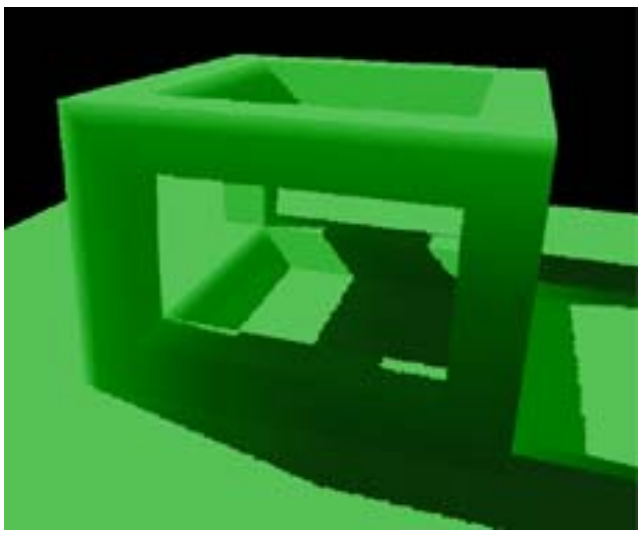
```
ps.1.3
def c7, 1.0, 0.66, 0.31, 0.0
def c6, -0.01, -0.01, -0.01, 0.0

tex t0                          // red+green ramp texture
tex t1                          // blue ramp texture
tex t3                          // depth of solid objs

add    t2,  t0, t1  // Add R + G + B to make depth value
add_x4 r1, -t3, t2  // r1 = diff * 4
add_x4 r1, r1, r1
add_x4 r1, r1, r1               // diff * 256, result is -1, 0, or 1 in each color
dp3_x4 r1, r1, c7               // weight R, G, B to make + or - value
// The sign of r1 reflects whether the value which t2 represents is greater
//  than or less than the value which t3 represents
add r1, r1, c6                  // CMP performs >= 0, so subtract a small value from r1
cmp r0, r1, t3, t2  // r1.rgb >= 0 ? t3.rgb : t2.rgb
```

Further Uses : Translucency

- Color ramp based on distance light travels through an object
- Similar to shadow maps



Greg James, NVIDIA

Simon Green, NVIDIA

Further Ideas

- Attenuation from volumes
  - Simulate light scattering or absorption
  - Darken things behind the volumes
- Turbulence texture
  - RGB-encoded turbulence applied in order to add and subtract thickness
  - Enhance simple volume fog geometry
  - Animate the texture
- Animate the volume objects

Additional Credits

- NVIDIA DevTech & DemoTech!
- Matthias Wloka
  - Neighbor sampling & convolution
- Gary King
  - Parallel development GeForce FX OGL volume fog
- Simon Green
  - Translucency, endless supply of cool articles!
- Microsoft
- Chas Boyd & co.
  - DXSDK examples

Begin Alex

Two more special effects

- **Utilizing Destination Alpha For Composite Shadows**
- **Advanced Fur Rendering Techniques**

Composite Shadows

A technique for combining pre-computed shadows with dynamic, stencil-based shadows

Goals of Composite Shadows

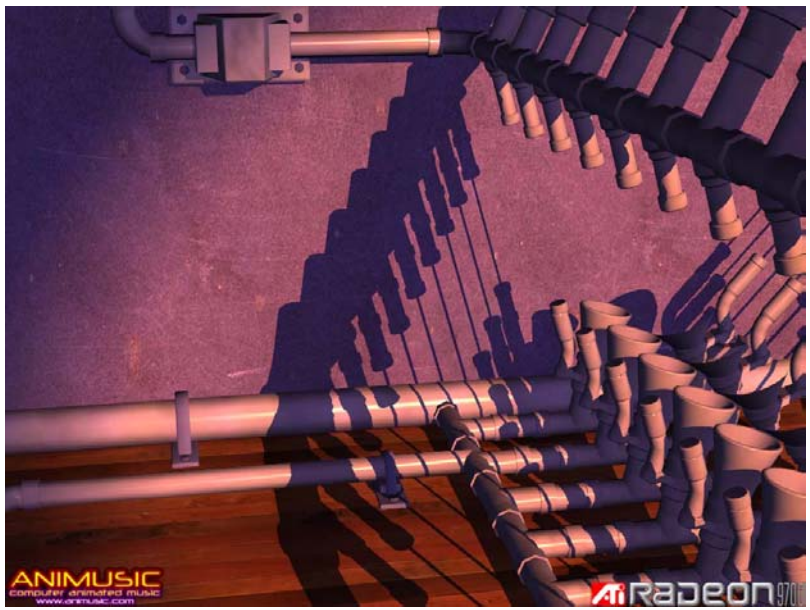
- Look like stencil shadow volumes globally
- Less expensive than global shadow volumes by not requiring a pass of your scene geometry per light
- Use shadow cutting to optimize static shadows (or another technique)
- Reserve shadow volumes for dynamic objects
- Use destination alpha for composite operation

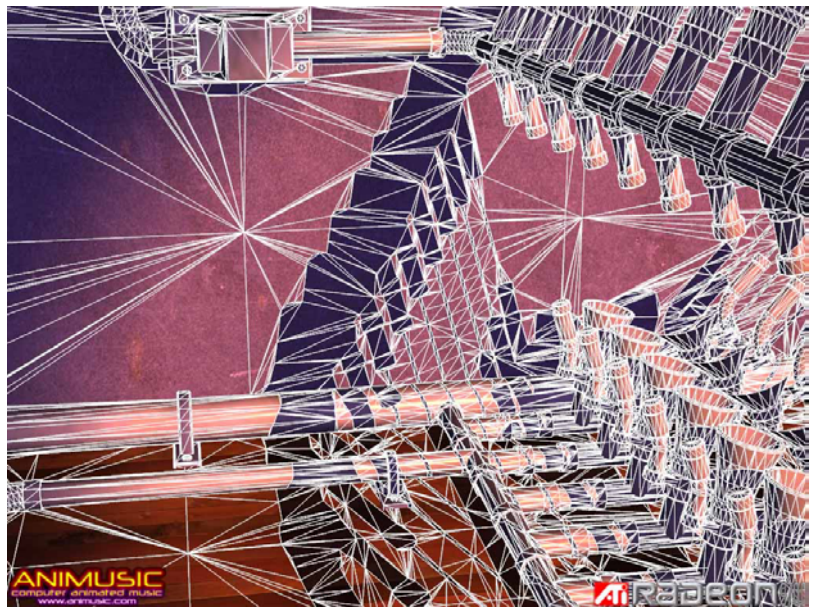
Static Shadows

- Shadows cast by non-moving light sources
- Scene geometry that doesn't move
  - Terrain, rocks, buildings, etc.
- Great opportunity to optimize out the brute-force nature of dynamic shadow volumes!

Static Shadows

- **Precomputed shadows are cut directly into the artist-generated geometry**



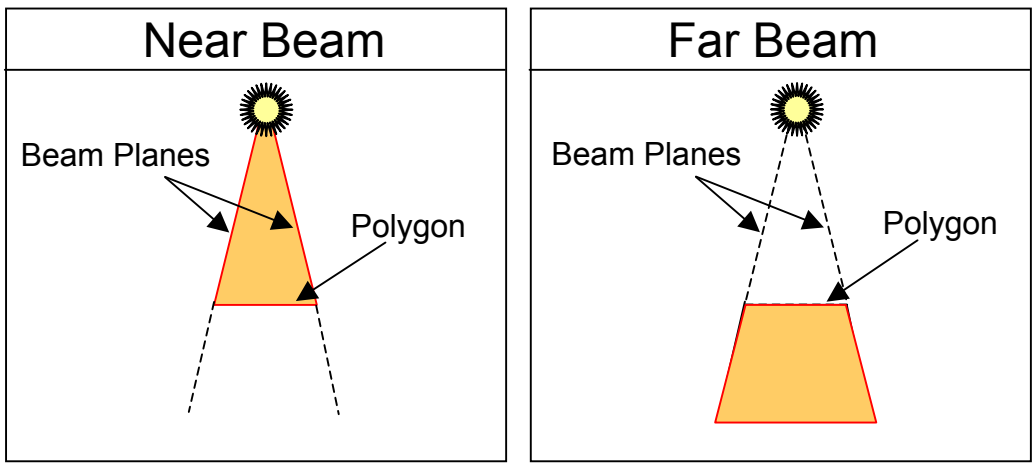


Advantages

- Looks like global stencil shadow volumes without the fill overhead!
- We can draw the polygons that are in shadow with a simpler vertex and pixel shader since fewer lights are affecting those pixels.
- The light color that casts the shadows are still animated! Dimming and color change is possible.

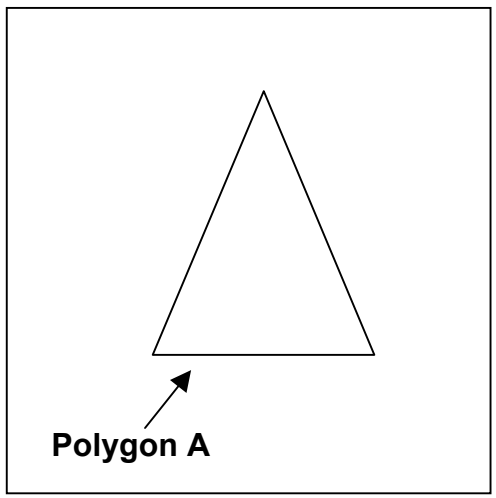
Beam Basics

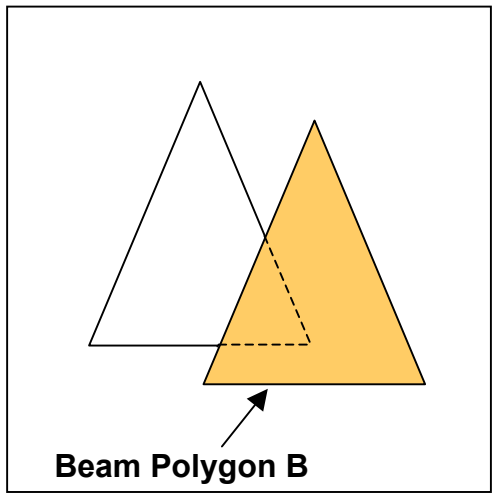
- A beam is a closed volume created by 4 planes.
- The 4 planes are constructed from a single triangle and a light's position.
- 3 of the 4 planes are defined by an edge of the polygon and the light's position.
- The 4th plane is simply the plane of the triangle.
- Near vs. Far beam: flip the normal of the 4th plane.

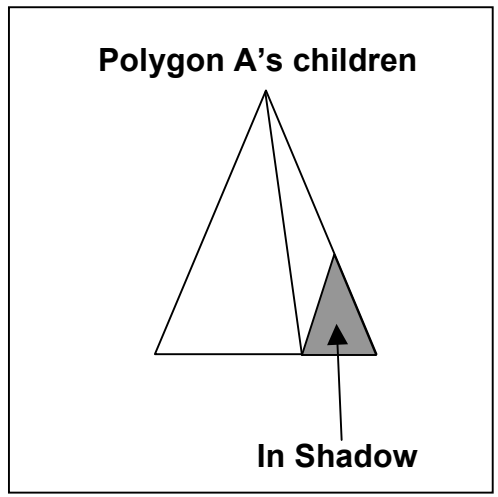


Shadow Cutting Algorithm

- For each polygon in scene (polygon A)
  - For each polygon that falls into polygon A's "near beam" (polygon B)
    - "Far beam" polygon B into polygon A and mark fragments of A that fall inside of beam B as in shadow.

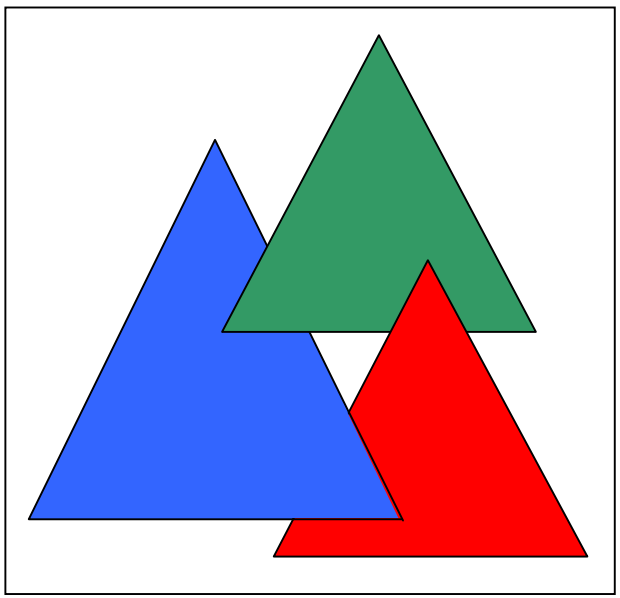


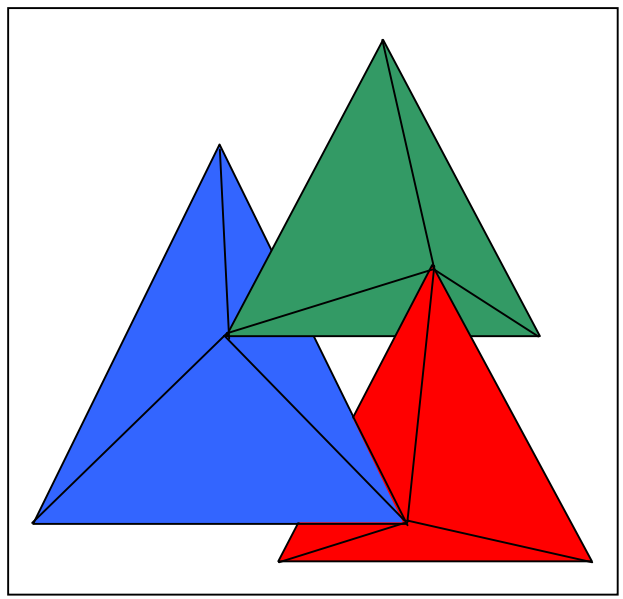




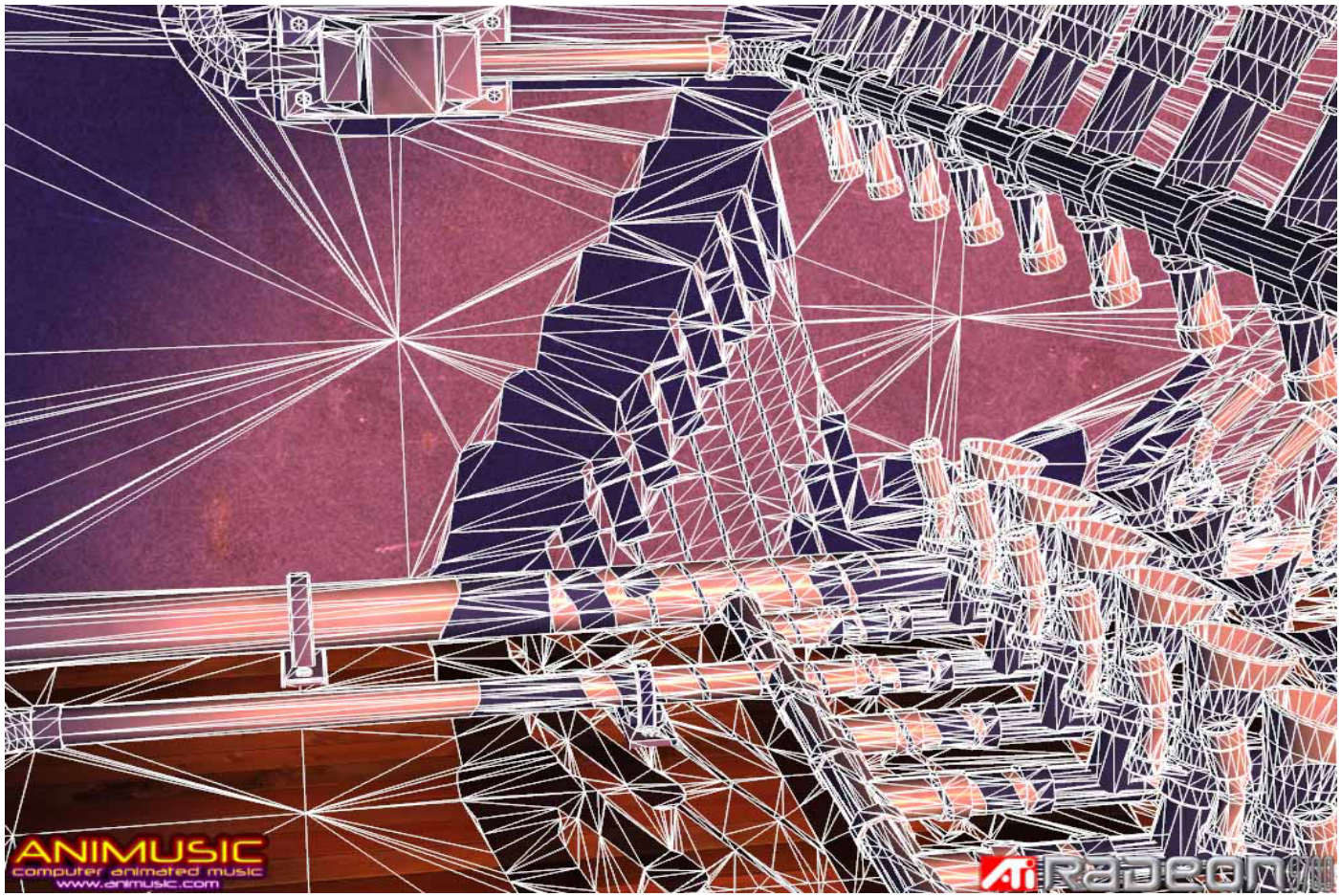
Difficult Cases

- Due to the simplicity of the algorithm, there is no recursion. Only original artist-created polygons form beams.
- This automatically solves for cyclically overlapping polygons without special code:



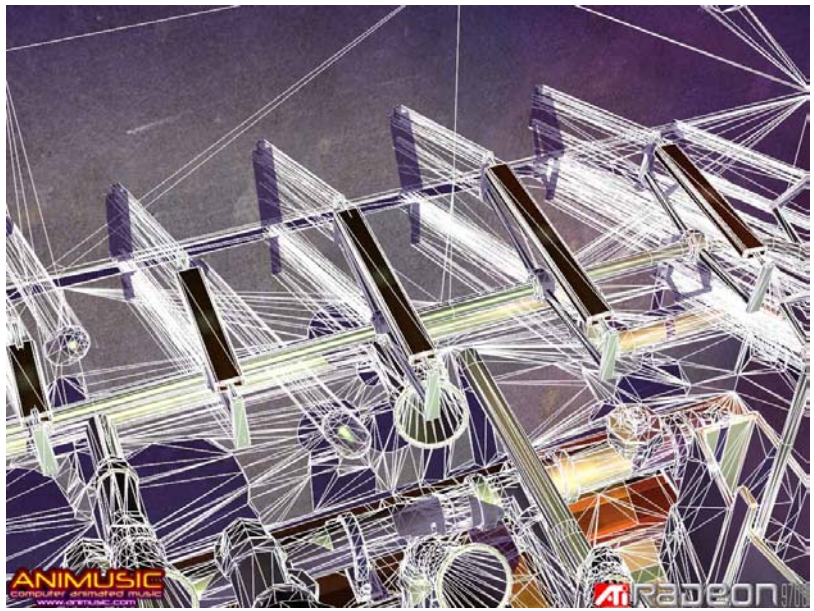


Result of Shadow Cutting

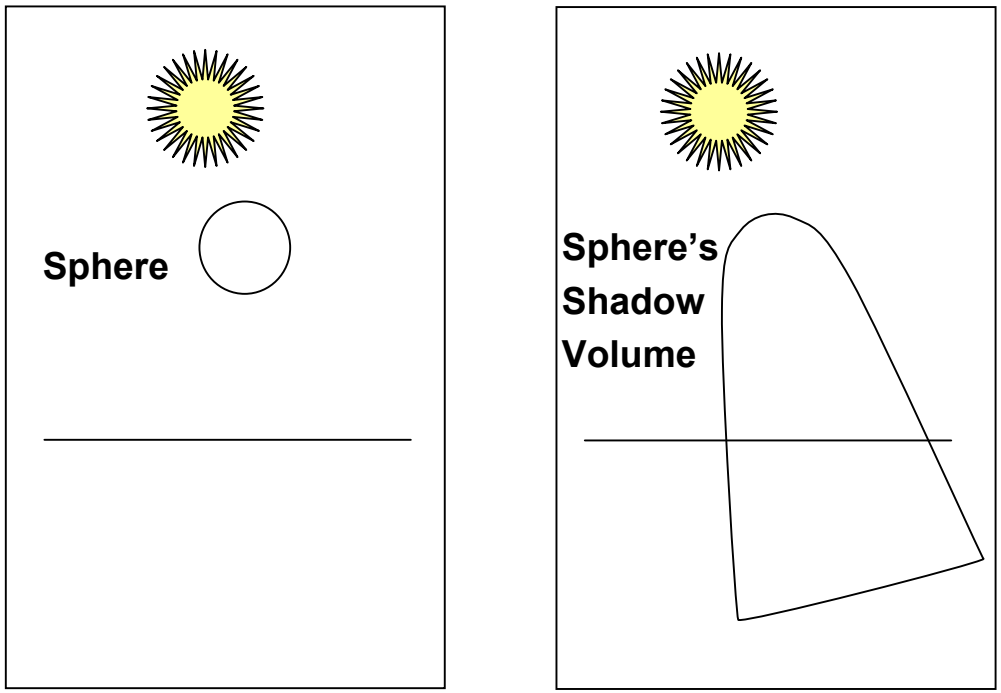


Dynamic Shadows

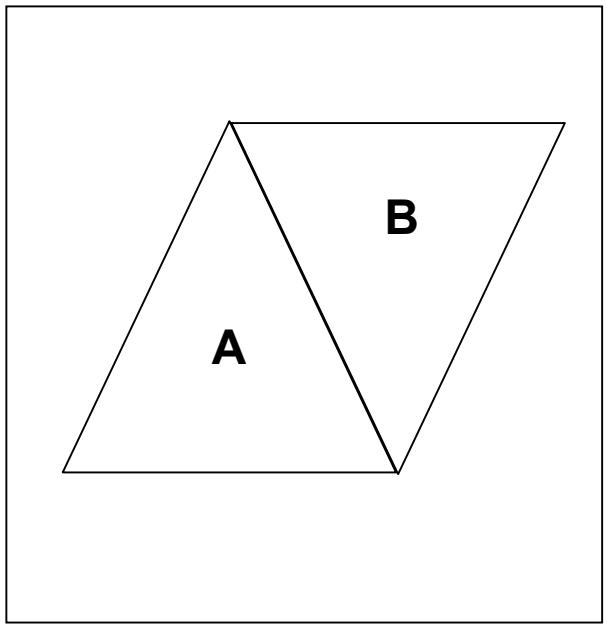
Dynamic shadows are used for animated geometry using stencil shadow volumes.

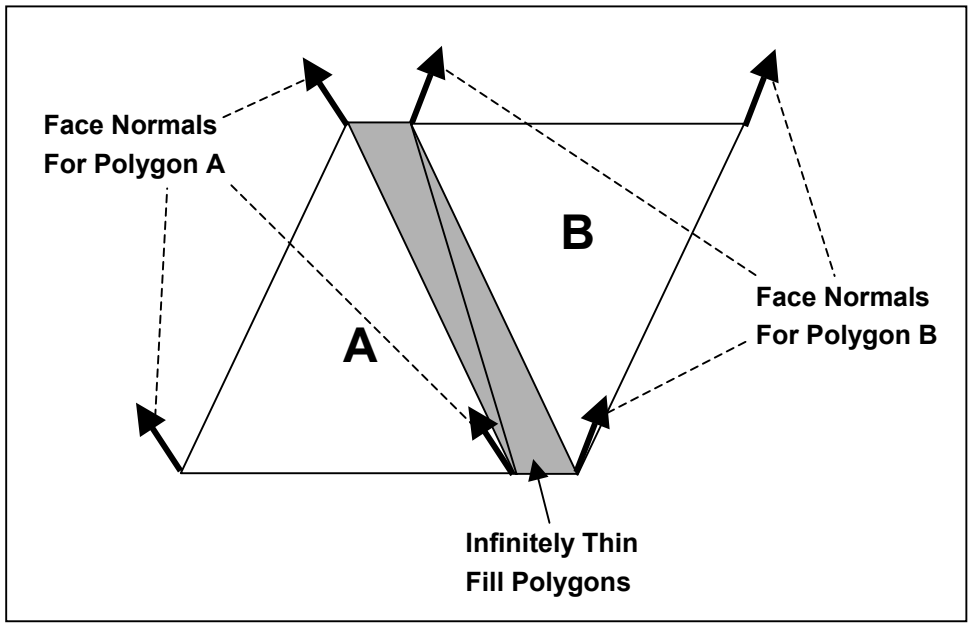


Shadow Volumes



Shadow Volume Extrusion Setup





Original bordering polygons.

We insert 2 degenerate polygons between the original polygons which share the appropriate face normal encoded in the vertex.

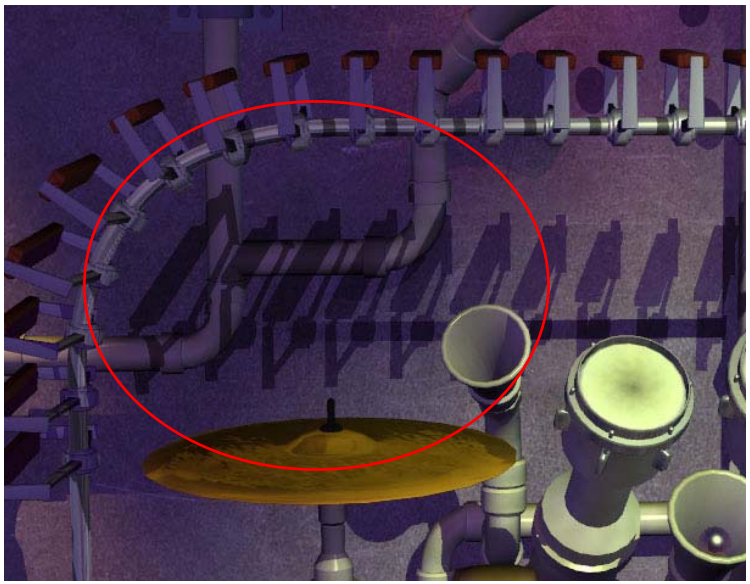
Stencil Buffer

- Dynamic shadow volumes are drawn into the stencil buffer.
- Pixels are essentially tagged as in or out of shadow based on their stencil value.

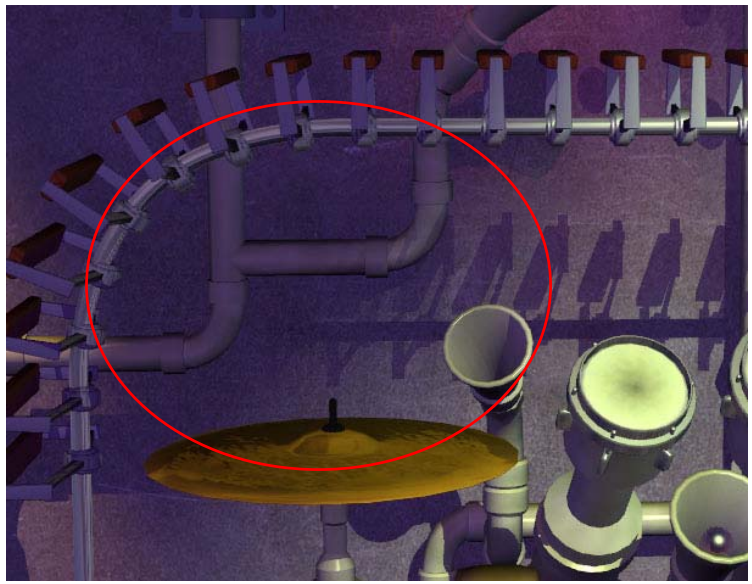
- How do we get the shadows into the color buffer?...

Masking The Correct Light

Brute-force stencil shadows without a per-pixel dimming factor causes harsh shadows and darkening below ambient light levels



Over Darkening From No Dimming



Per-Pixel Dimming Factor

Per-Pixel Dim Factor

- As the scene is drawn initially, we also write a useful value to destination alpha
- This grayscale value represents an approximation of how dark the pixel should be if it falls into shadow later
- Future work: use an RGB renderable texture for better color preservation

Per-Pixel Dim Factor in Dest Alpha

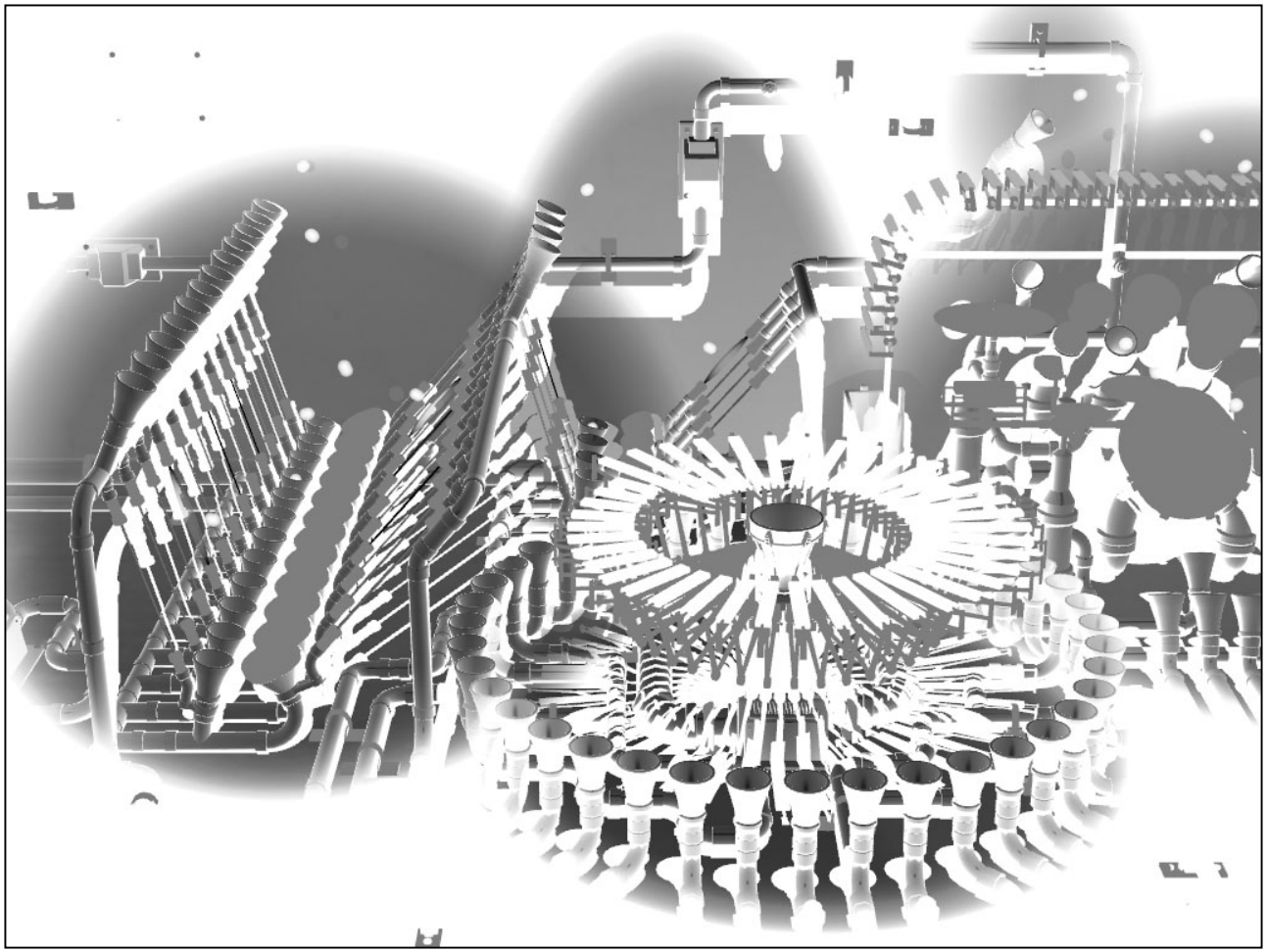
- Later, the dest alpha value will be multiplied by the dest color value to obtain the shadow color (ambient in this case)

- The dest alpha value solves the following equation per-pixel:

  destColor * destAlpha = ambient

  destAlpha = ambient / destColor

Contents of Destination Alpha



Composite Shadow Quad

- **A full-screen quad is drawn with the following state enabled:**
  - D3DRS_SRCBLEND = D3DBLEND_ZERO
  - D3DRS_DESTBLEND = D3DBLEND_DESTALPHA
  - Stencil state = Allow drawing only to pixels in shadow shadow

- **This provides:**

  (DestColor*DestAlpha) Masked By Stencil

Results of Composite Shadows

Advanced Fur Techniques

Fur Basics – Shells & Fins

&

=

Shells

Base geometry is grown in the direction of the vertex normal for each shell.

Fins

Two triangles are extruded along the direction of the edge normal for each edge.

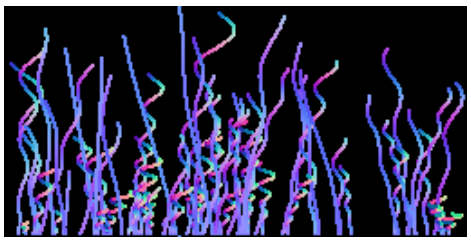
Shells & Fins

Shells and fins are combined.

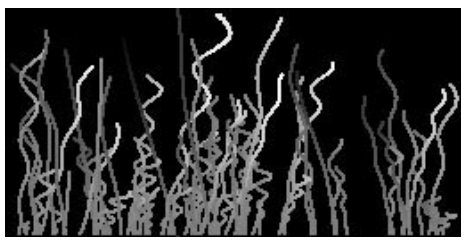
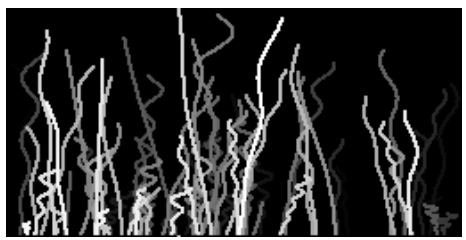
Fin Textures

Fins rendered with two RGBA textures

- RGB(DirOfAniso) & Alpha(Opacity)

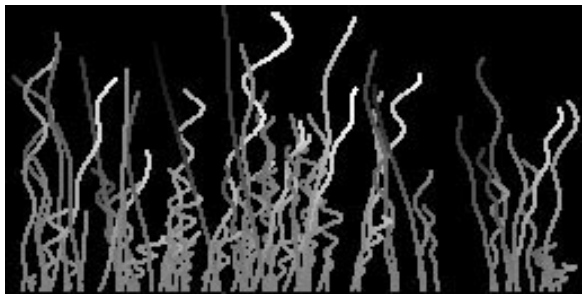



- R(uOffset), G(0), B(Thinning), & Alpha(LengthCulling)

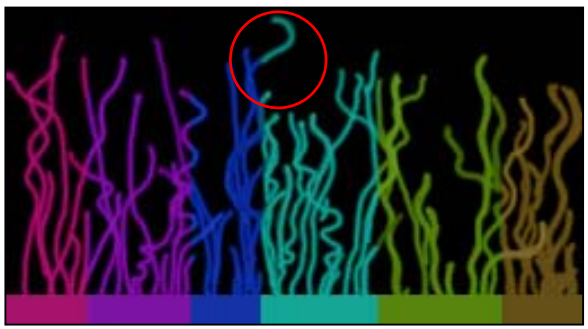




Fur Color Offset

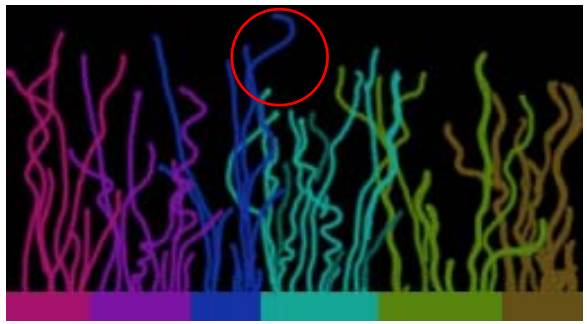
Fin offset texture used to color each strand of hair uniformly



Incorrect method of coloring hair – Stretching the color straight up

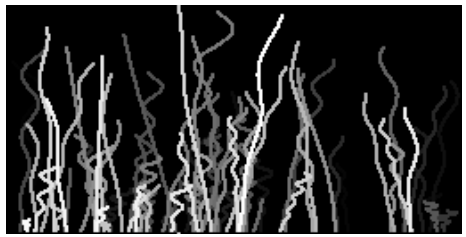


Correct method utilizing the offset texture to fetch the color of the hair from its base



Fur Thinning

Fin thinning texture represents the offset in texture coordinates to fetch from the base of the strand of fur



Threshold on the fur thinning texture:

0

64

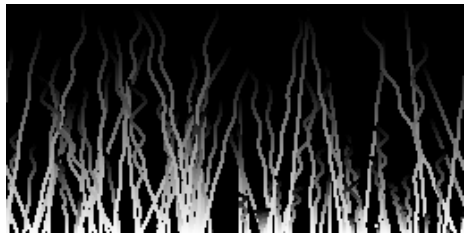
128

192

245

Fur Length Culling

Fin length culling texture contains normalized fur length per-strand



Ugly length by chopping off image:

Threshold on the fur length culling texture:

0 64 128 192 245

Bald Spots

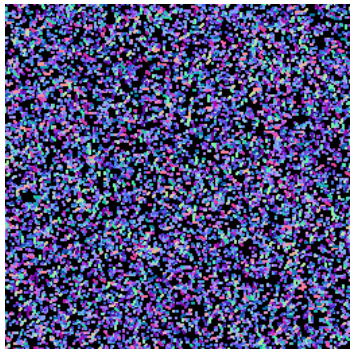
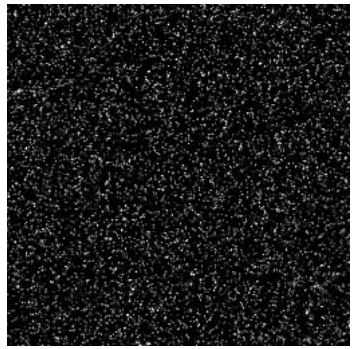
Real chimps have bald spots on their foreheads!

We utilize length culling and thinning to replicate this.

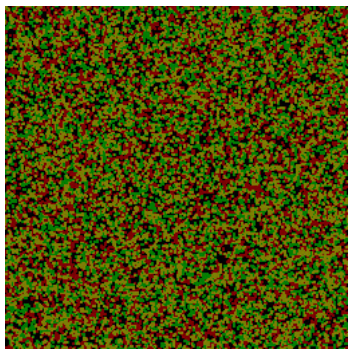
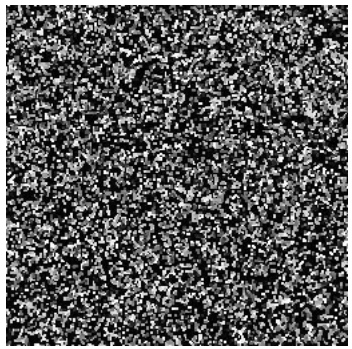
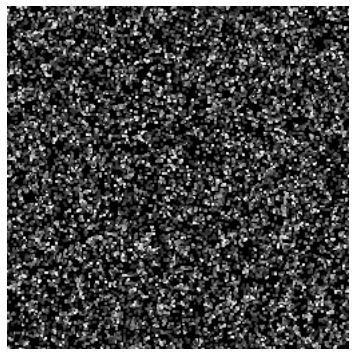
Shell Textures

Shells rendered with two RGBA textures

- RGB(DirOfAniso) & Alpha(Opacity)

- RG(uvOffset), B(Thinning), & Alpha(LengthCulling)

Summary

- Glow effect
  - Developed for Disney/Monolith's "Tron 2.0"
- Volume fog from polygon objects
  - Used in Bandai/Dimps "UniversalCentury.net Gundam Online"
- Used destination alpha to blend pre-computed and dynamic shadows
  - Calculating the per-pixel dim factor
  - Direct3D blend state for final fullscreen quad
- Advanced fur techniques
  - Color offset
  - Fur thinning
  - Fur length culling

Acknowledgements

- **ATI 3D Application Research Group Demo Team**

Questions?

Where are the slides?

- www.ati.com/developer
- www.nvidia.com/developer

- **We'll post them in the next few days**