



Shadow Mapping with Today's OpenGL Hardware

**CEDEC 2001
Tokyo, Japan**

Mark J. Kilgard
Graphics Software Engineer
NVIDIA Corporation



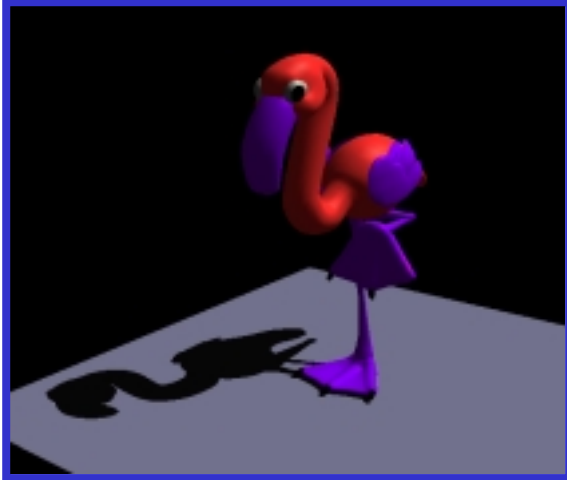
*n*VIDIA™

Motivation for Better Shadows

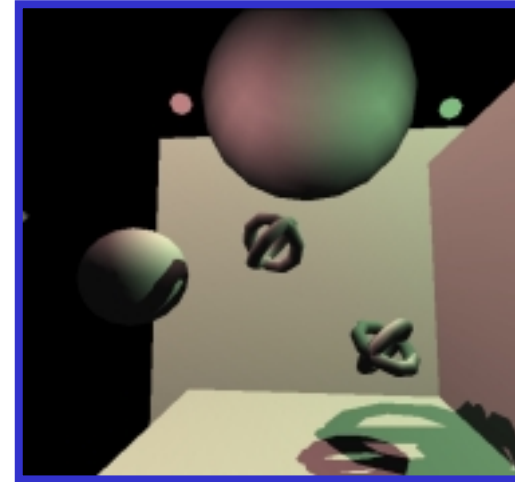
- **Shadows increase scene realism**
 - Real world has shadows
 - More control of the game's feel
 - dramatic effects
 - spooky effects
 - Other art forms recognize the value of shadows
 - But yet most games lack realistic shadows



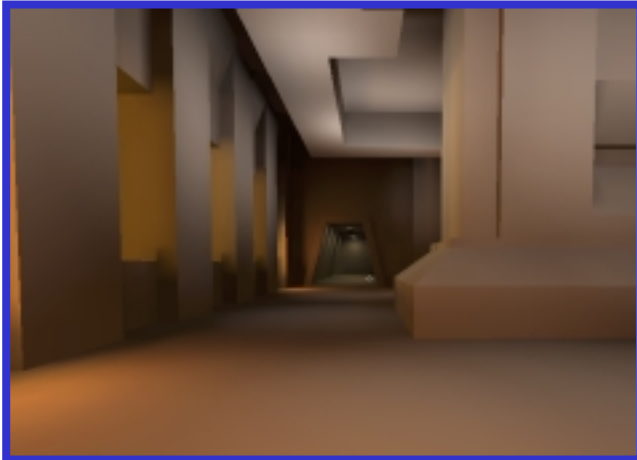
Common Real-time Shadow Techniques



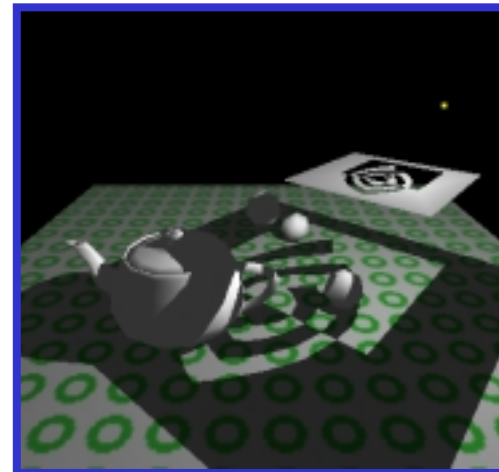
*Projected
planar
shadows*



*Shadow
volumes*



Light maps



*Hybrid
approaches*



nVIDIA™

Problems with Common Shadow Techniques

- **Mostly tricks with lots of limitations**
 - **Projected planar shadows**
 - **well works only on flat surfaces**
 - **Stenciled shadow volumes**
 - **determining the shadow volume is hard work**
 - **Light maps**
 - **totally unsuited for dynamic shadows**
 - **In general, hard to get everything shadowing everything**



Introducing Another Technique: Shadow Mapping

- **Image-space shadow determination**
 - Lance Williams published the basic idea in 1978
 - By coincidence, same year Jim Blinn invented bump mapping (a great vintage year for graphics)
- **Completely image-space algorithm**
 - means no knowledge of scene's geometry is required
 - must deal with aliasing artifacts
- **Well known software rendering technique**
 - Pixar's RenderMan uses the algorithm
 - Basic shadowing technique for Toy Story, etc.



Shadow Mapping References

- **Important SIGGRAPH papers**
 - Lance Williams, “Casting Curved Shadows on Curved Surfaces,” SIGGRAPH 78
 - William Reeves, David Salesin, and Robert Cook (Pixar), “Rendering antialiased shadows with depth maps,” SIGGRAPH 87
 - Mark Segal, et. al. (SGI), “Fast Shadows and Lighting Effects Using Texture Mapping,” SIGGRAPH 92



The Shadow Mapping Concept (1)

- **Depth testing from the light's point-of-view**
 - Two pass algorithm
 - First, render depth buffer from the light's point-of-view
 - the result is a “depth map” or “shadow map”
 - essentially a 2D function indicating the depth of the closest pixels to the light
 - This depth map is used in the second pass



The Shadow Mapping Concept (2)

- **Shadow determination with the depth map**
 - **Second, render scene from the eye's point-of-view**
 - **For each rasterized fragment**
 - **determine fragment's XYZ position relative to the light**
 - **this light position should be setup to match the frustum used to create the depth map**
 - **compare the depth value at light position XY in the depth map to fragment's light position Z**



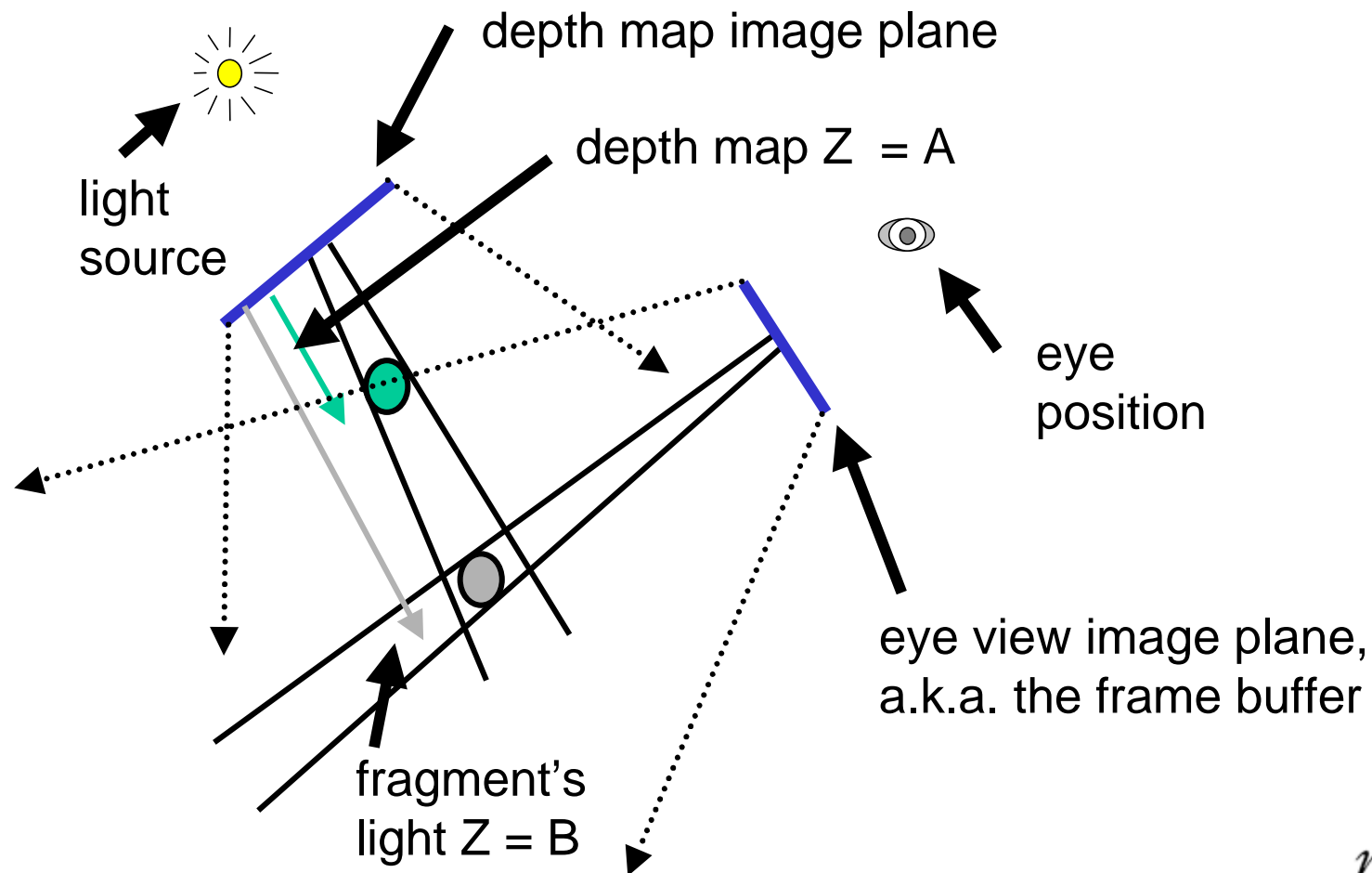
The Shadow Mapping Concept (3)

- **The Shadow Map Comparison**
 - **Two values**
 - **A = Z value from depth map at fragment's light XY position**
 - **B = Z value of fragment's XYZ light position**
 - **If B is greater than A, then there must be something closer to the light than the fragment**
 - **then the fragment is shadowed**
 - **If A and B are approximately equal, the fragment is lit**



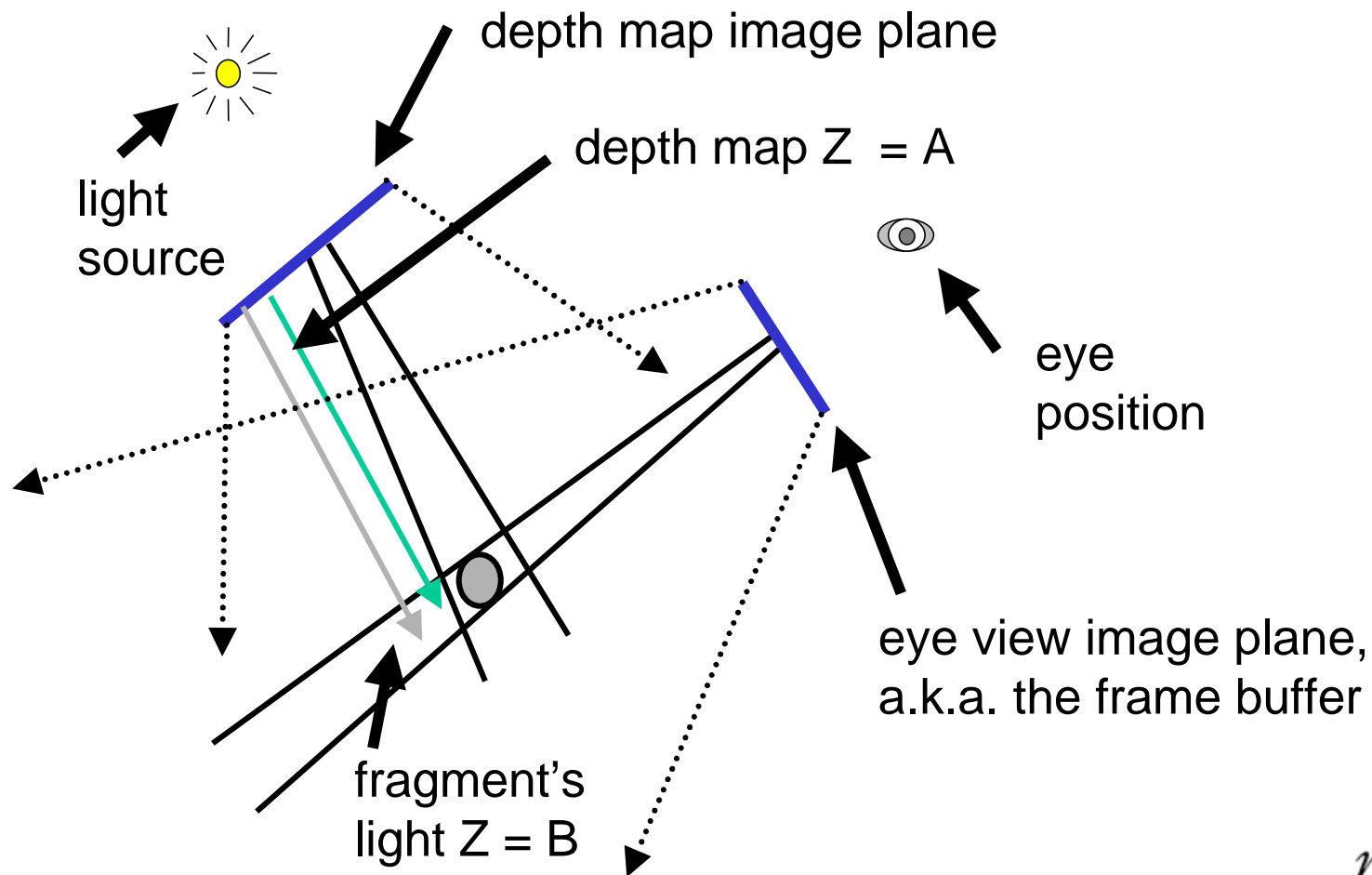
Shadow Mapping with a Picture in 2D (1)

The $A < B$ shadowed fragment case



Shadow Mapping with a Picture in 2D (2)

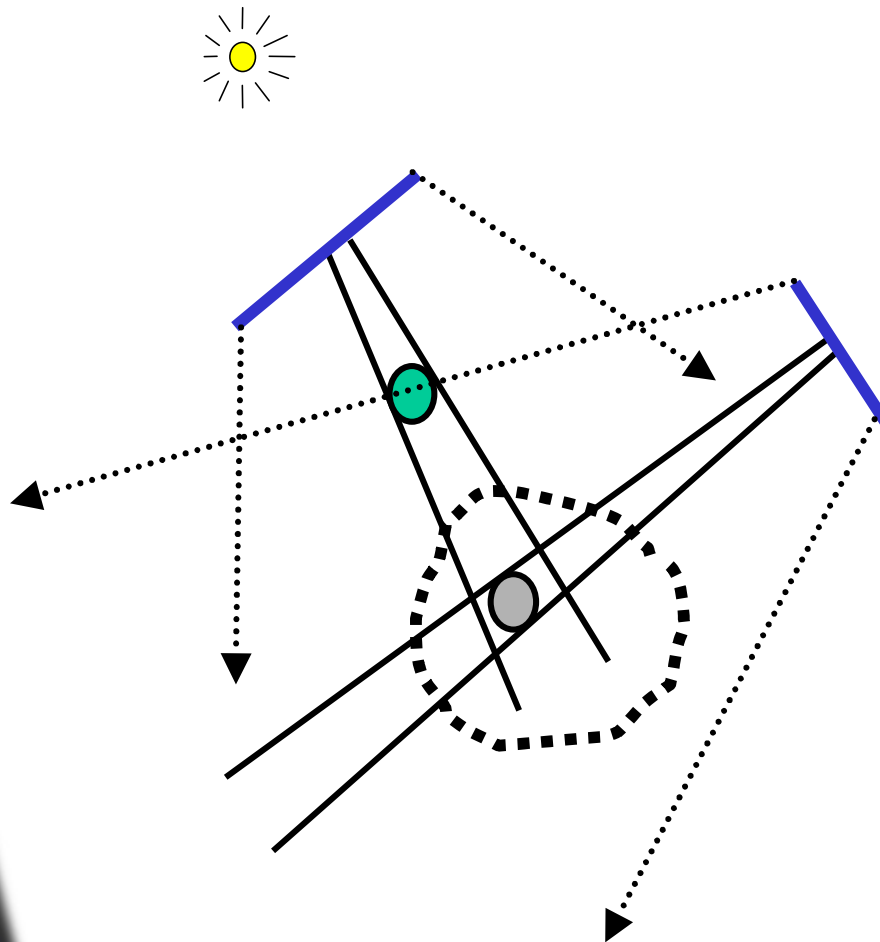
The $A \cong B$ unshadowed fragment case



nVIDIA

Shadow Mapping with a Picture in 2D (3)

Note image precision mismatch!



The depth map
could be at a
different resolution
from the framebuffer

This mismatch can
lead to artifacts

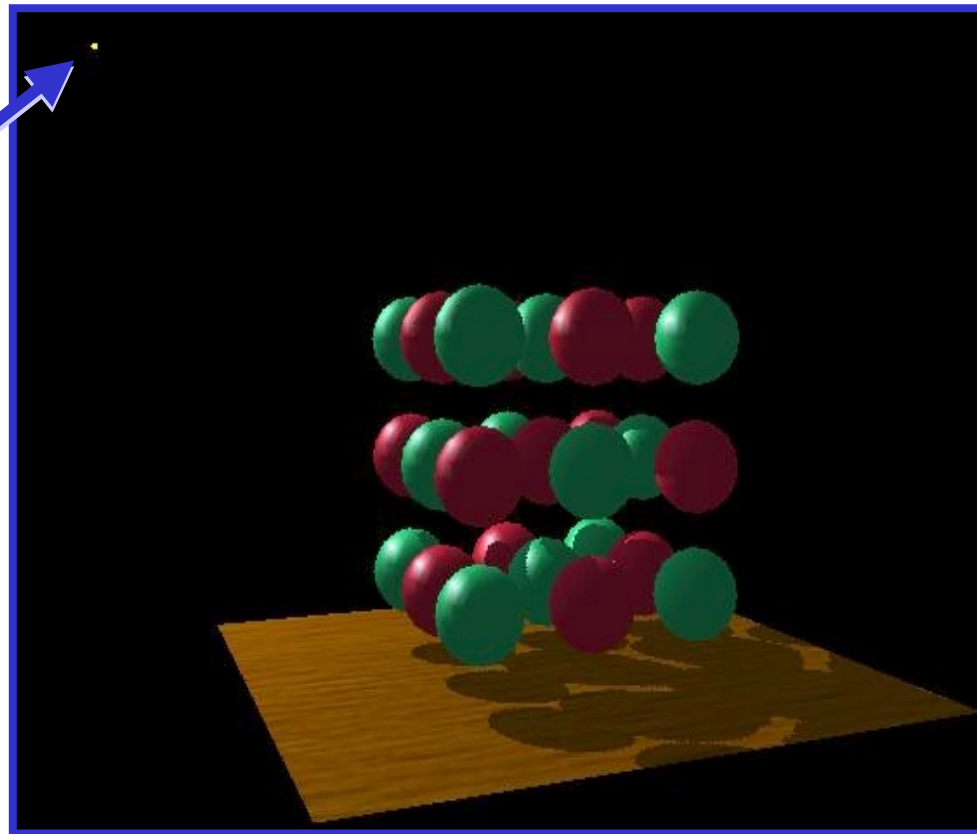


NVIDIA

Visualizing the Shadow Mapping Technique (1)

- A fairly complex scene with shadows

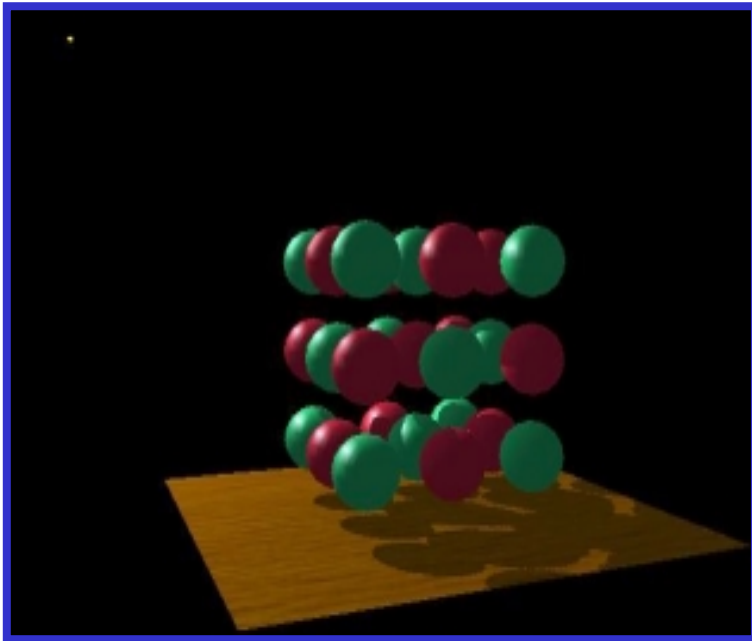
*the point
light source*



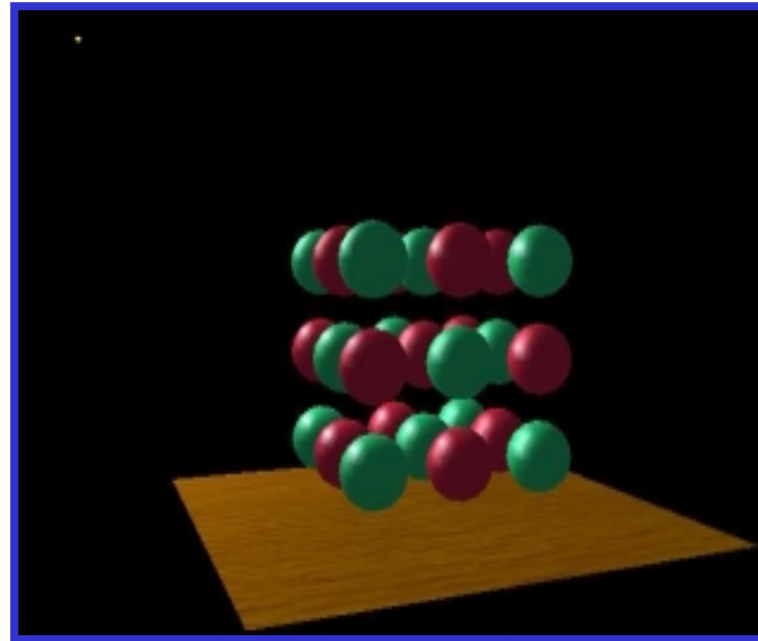
nVIDIA™

Visualizing the Shadow Mapping Technique (2)

- Compare with and without shadows



with shadows



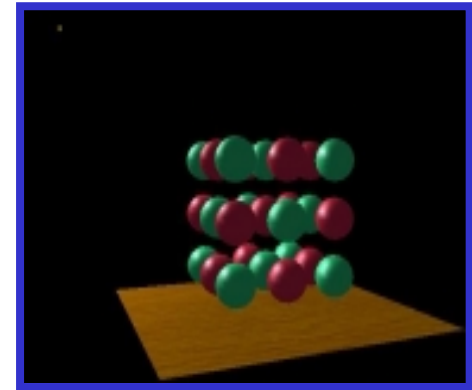
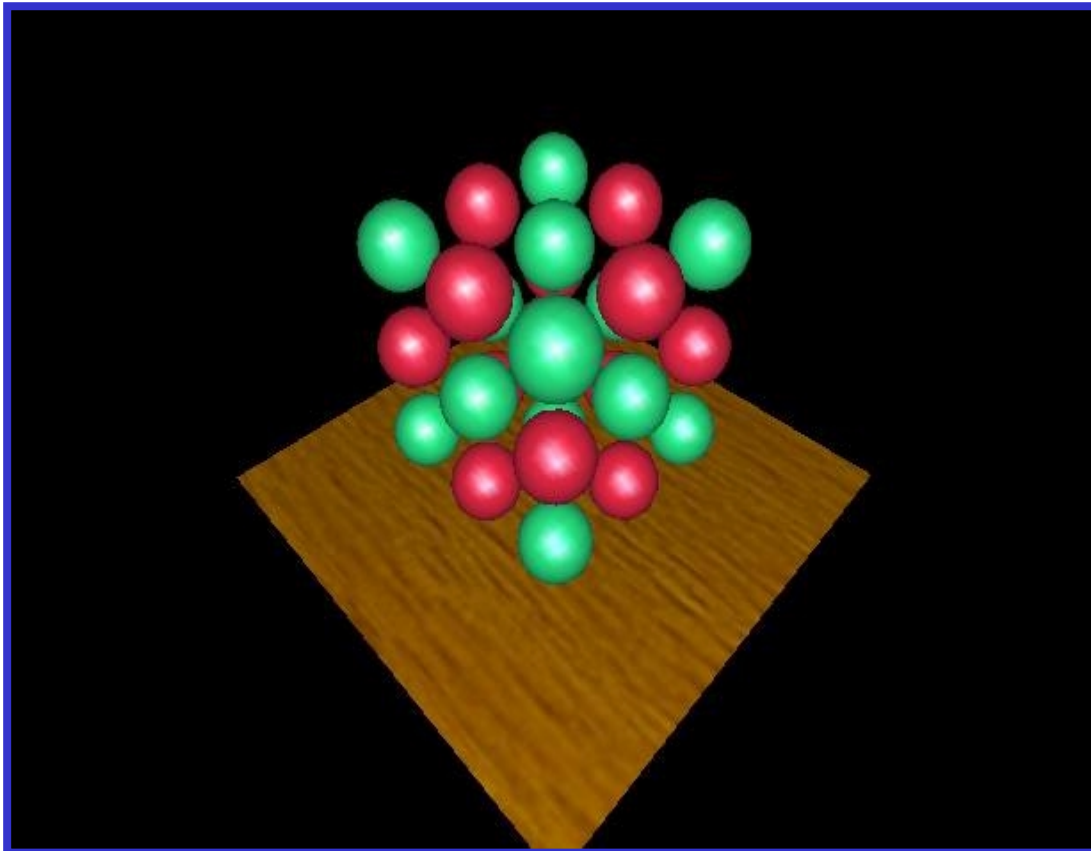
without shadows



NVIDIA™

Visualizing the Shadow Mapping Technique (3)

- The scene from the light's point-of-view



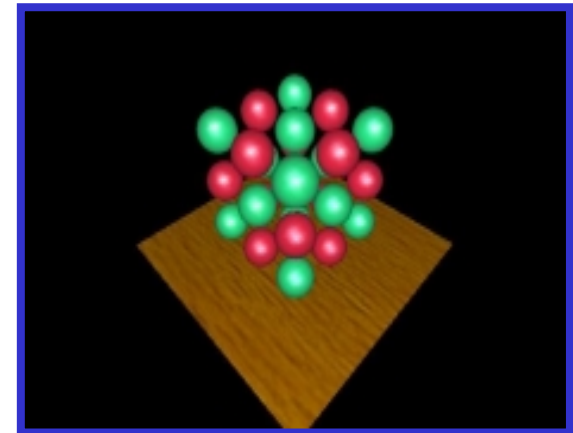
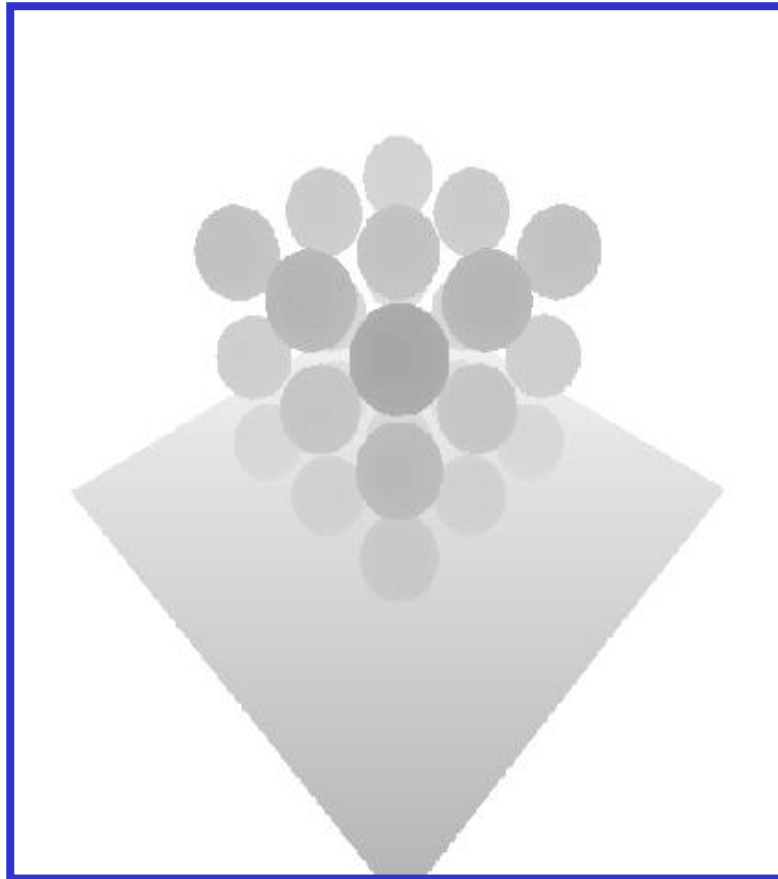
FYI: from the eye's point-of-view again



nVIDIA™

Visualizing the Shadow Mapping Technique (4)

- The depth buffer from the light's point-of-view



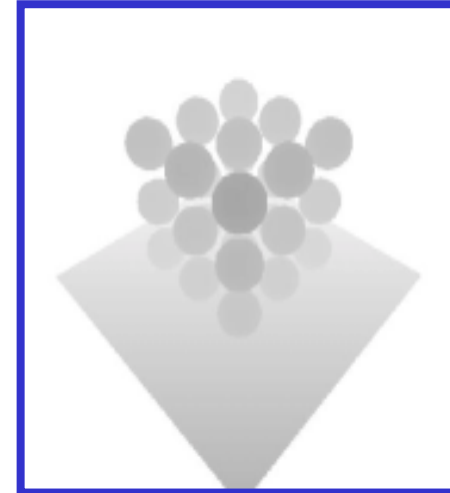
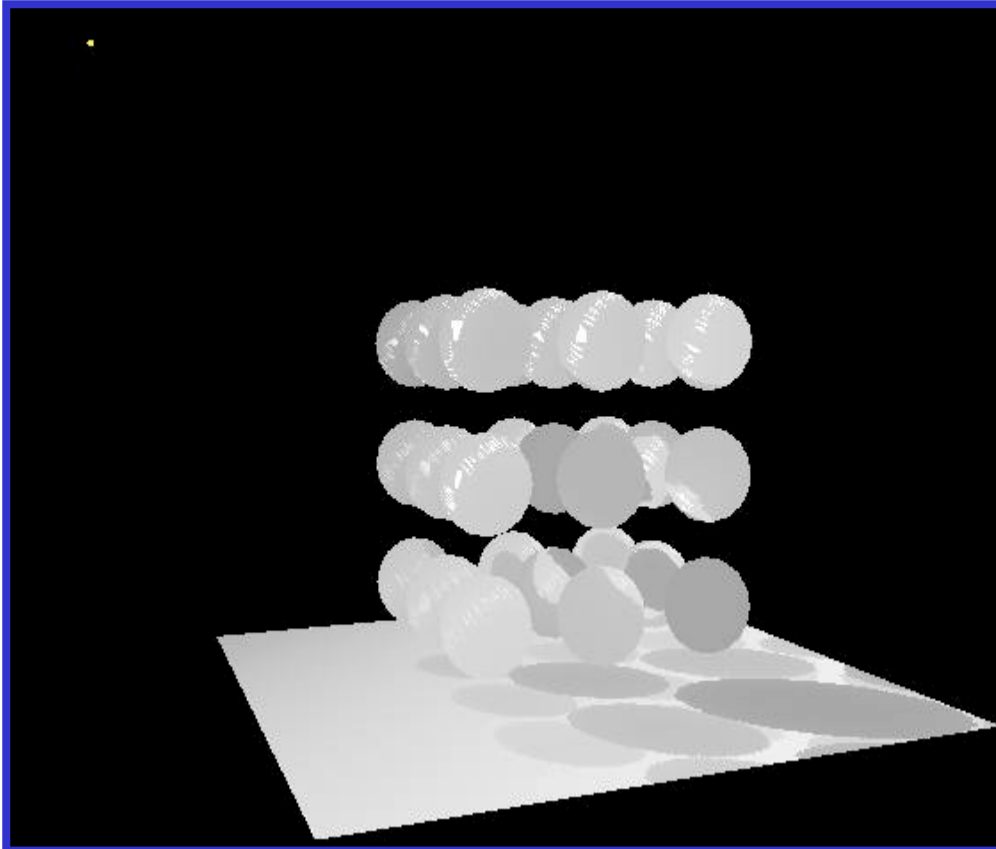
***FYI: from the
light's point-of-view
again***



nVIDIA™

Visualizing the Shadow Mapping Technique (5)

- Projecting the depth map onto the eye's view



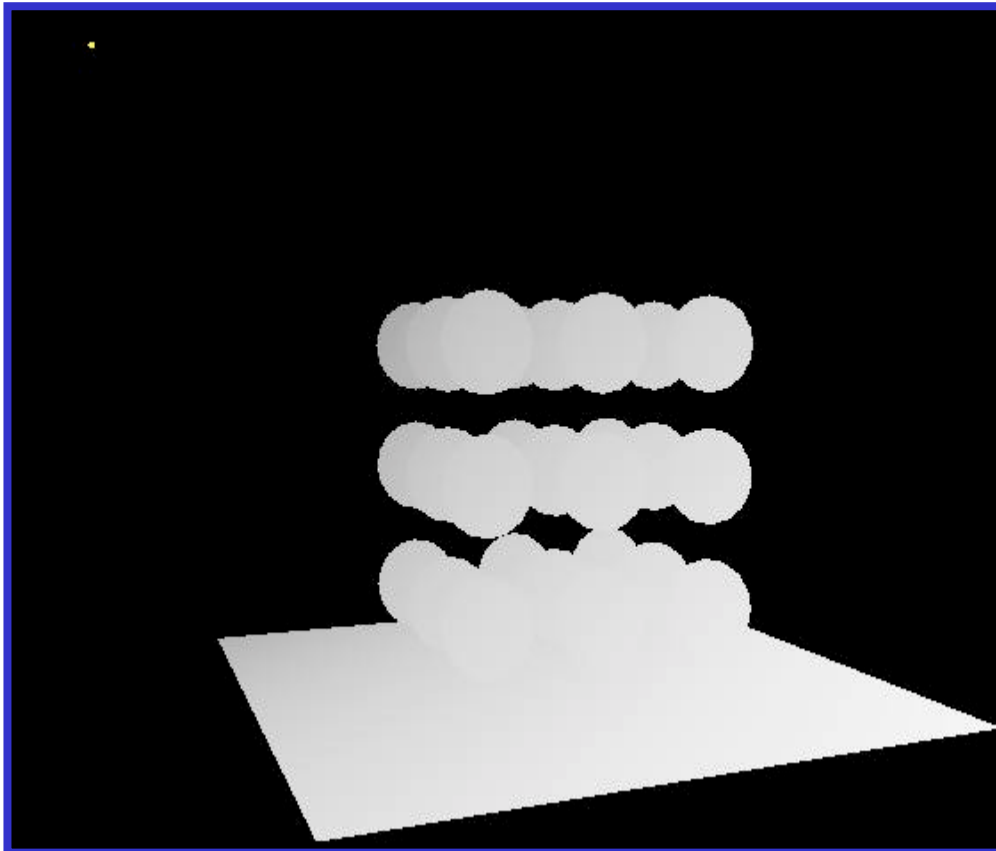
***FYI: depth map for
light's point-of-view
again***



nVIDIA™

Visualizing the Shadow Mapping Technique (6)

- Projecting light's planar distance onto eye's view



NVIDIA™

Visualizing the Shadow Mapping Technique (6)

- Comparing light distance to light depth map

*Green is
where the
light planar
distance and
the light
depth map
are
approximatel
y equal*



*Non-green is
where
shadows
should be*

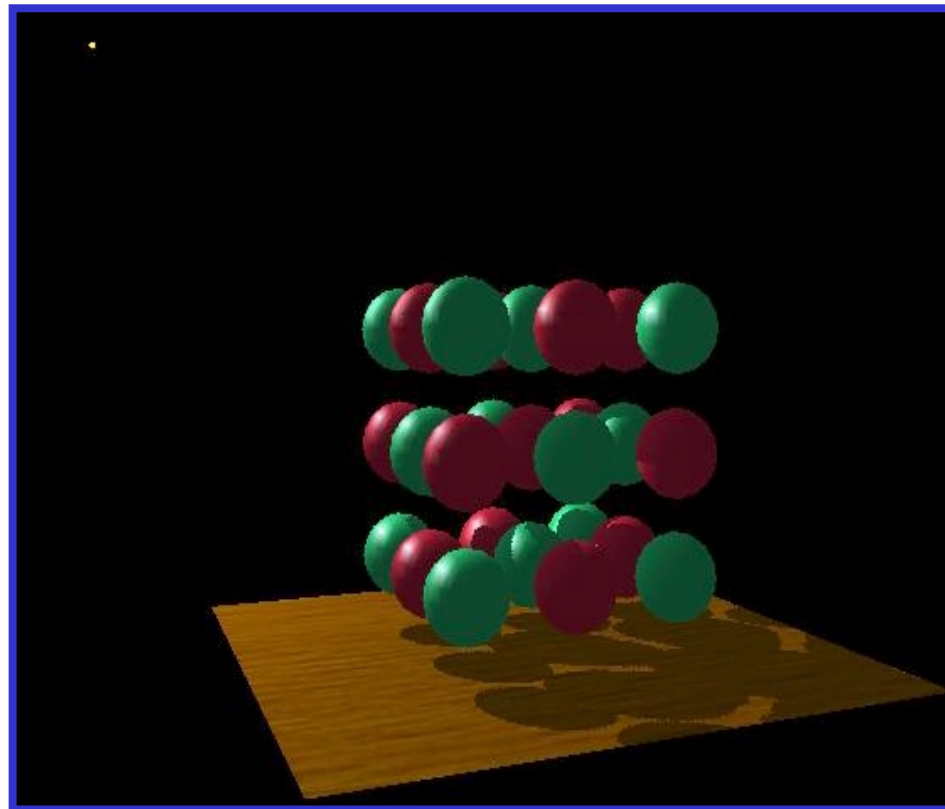


NVIDIA™

Visualizing the Shadow Mapping Technique (7)

- Scene with shadows

*Notice how
specular
highlights
never appear
in shadows*



*Notice how
curved
surfaces cast
shadows on
each other*



nVIDIA™

Construct Light View Depth Map

- **Realizing the theory in practice**
 - **Constructing the depth map**
 - **use existing hardware depth buffer**
 - **use glPolygonOffset to offset depth value back**
 - **read back the depth buffer contents**
 - **Depth map can be copied to a 2D texture**
 - **unfortunately, depth values tend to require more precision than 8-bit typical for textures**
 - **depth precision typically 16-bit or 24-bit**



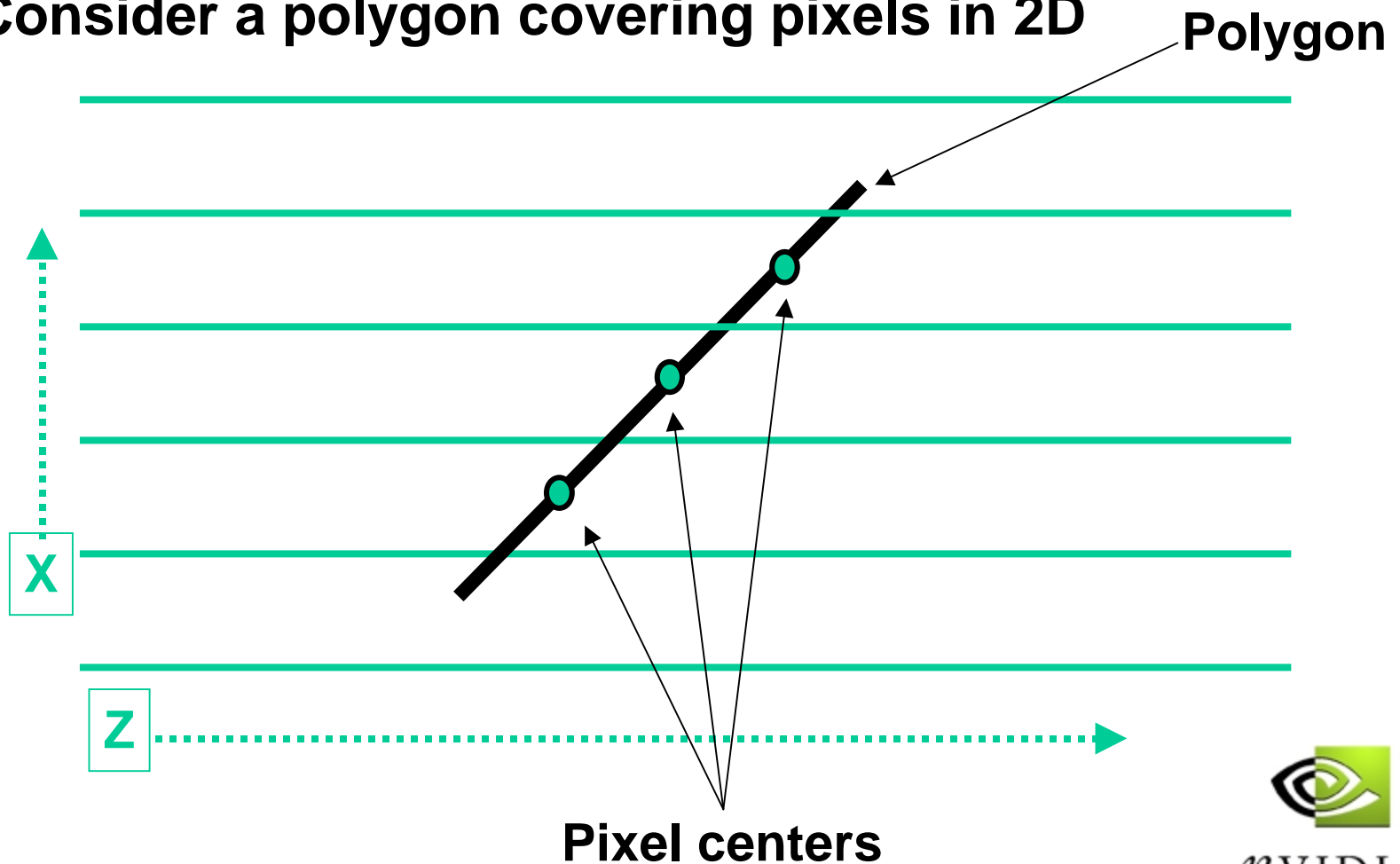
Justification for glPolygonOffset When Constructing Shadow Maps

- Depth buffer contains “window space” depth values
 - Post-perspective divide means non-linear distribution
 - glPolygonOffset is guaranteed to be a window space offset
- Doing a “clip space” glTranslatef is not sufficient
 - Common shadow mapping implementation mistake
 - Actual bias in depth buffer units will vary over the frustum
 - No way to account for slope of polygon



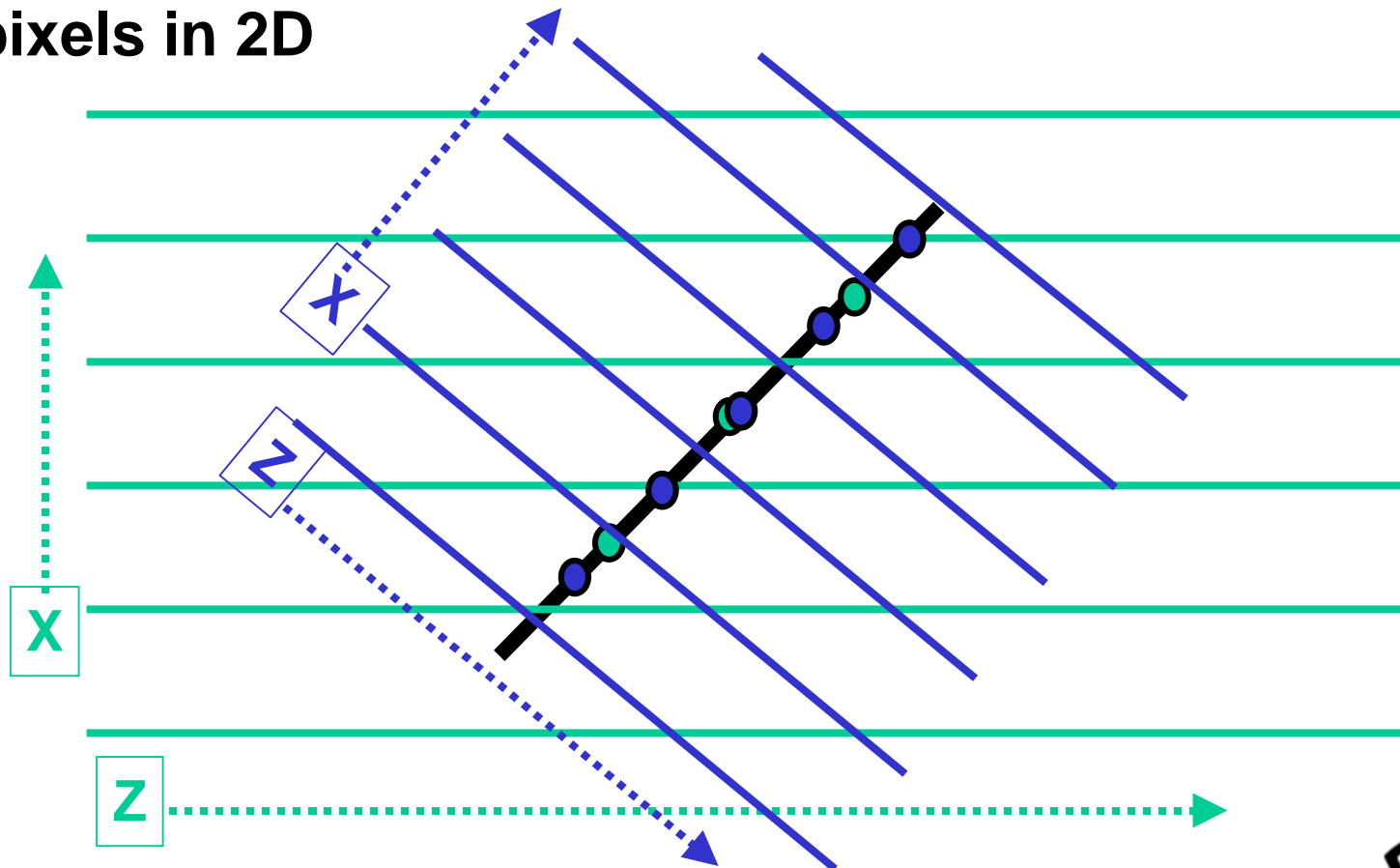
Sampling a Polygon's Depth at Pixel Centers (1)

- Consider a polygon covering pixels in 2D



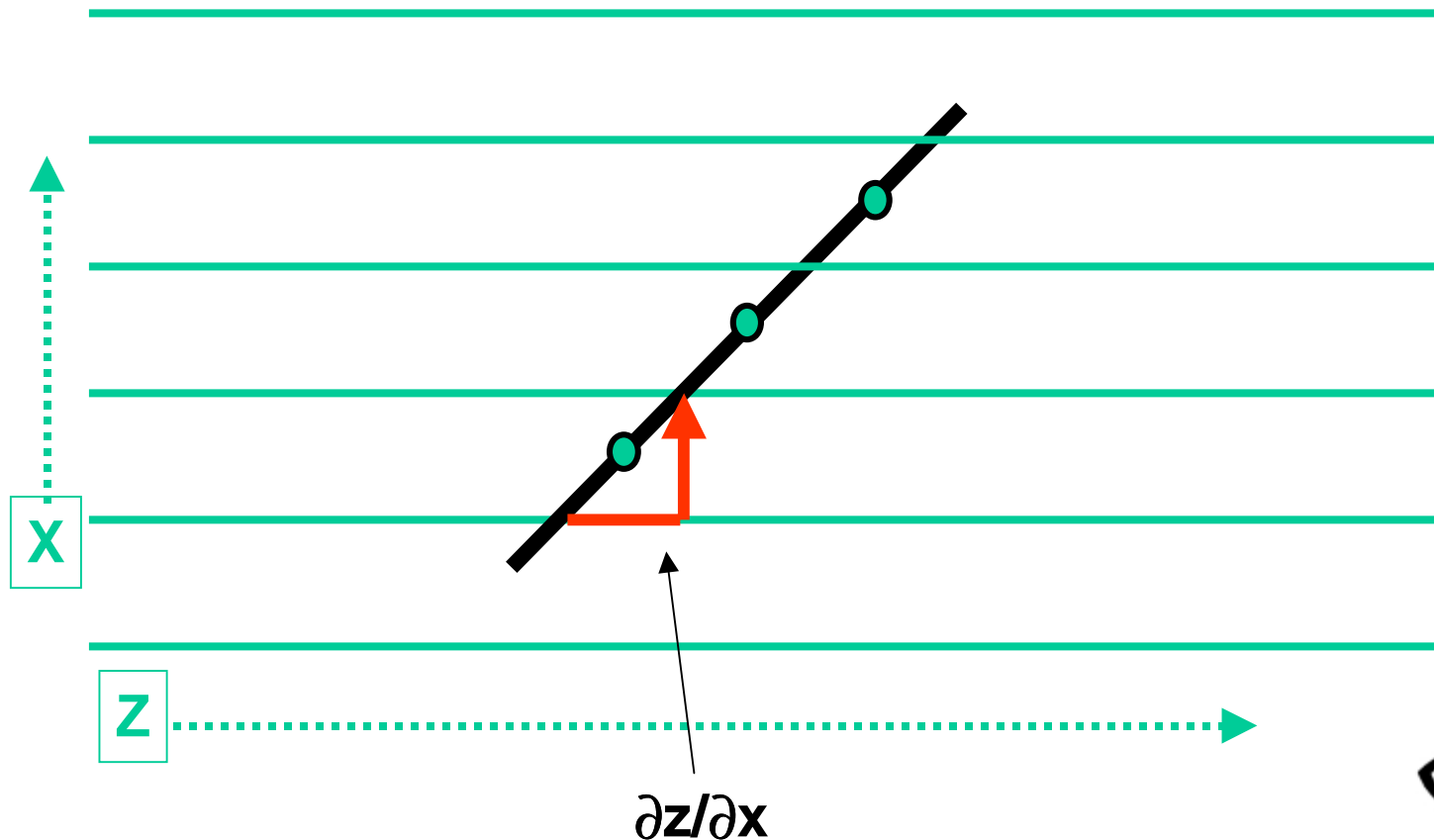
Sampling a Polygon's Depth at Pixel Centers (2)

- Consider a 2nd grid for the polygon covering pixels in 2D



Sampling a Polygon's Depth at Pixel Centers (3)

- How Z changes with respect to X



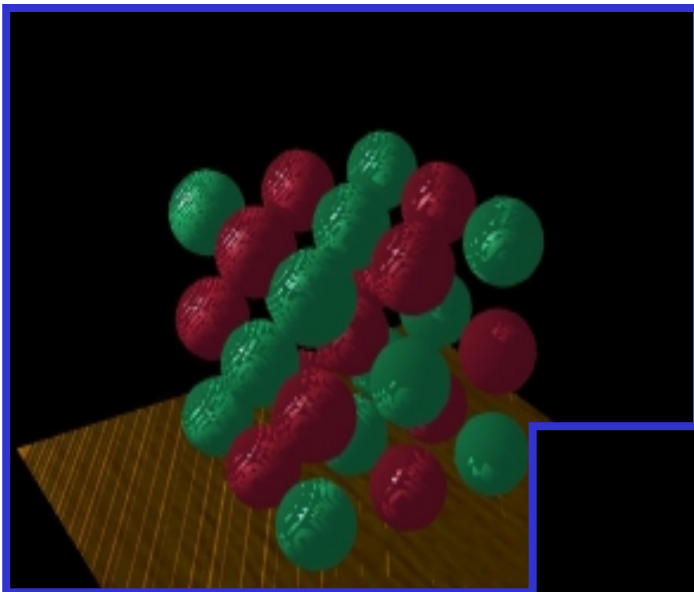
Why You Need glPolygonOffset's Slope

- Say pixel center is re-sampled to another grid
 - For example, the shadow map texture's grid!
- The re-sampled depth could be off by $\pm 0.5 \frac{\partial z}{\partial x}$ and $\pm 0.5 \frac{\partial z}{\partial y}$
- The maximum absolute error would be $|0.5 \frac{\partial z}{\partial x}| + |0.5 \frac{\partial z}{\partial y}| \approx \max(|\frac{\partial z}{\partial x}|, |\frac{\partial z}{\partial y}|)$
 - This assumes the two grids have pixel footprint area ratios of 1.0
 - Otherwise, we might need to scale by the ratio
- Exactly what polygon offset's "slope" depth bias does



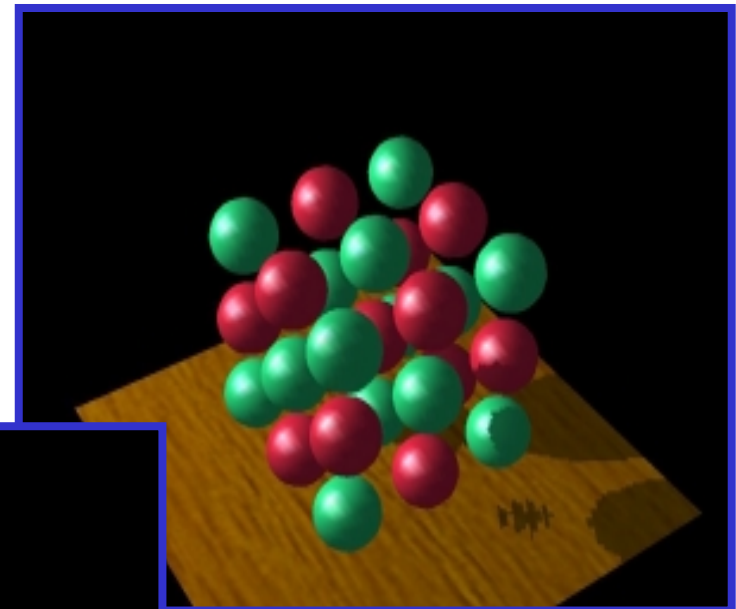
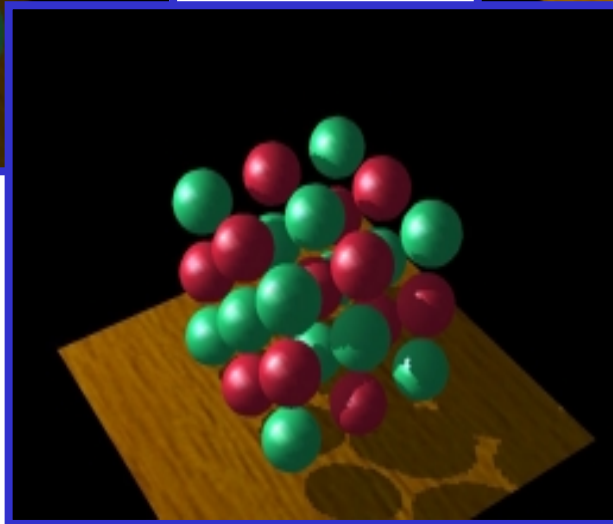
Depth Map Bias Issues

- How much polygon offset bias depends



*Too little bias,
everything begins to
shadow*

Just right



*Too much bias, shadow
starts too far back*



nVIDIA™

Selecting the Depth Map Bias

- **Not that hard**
 - **Usually the following works well**
 - **`glPolygonOffset(scale = 1.1, bias = 4.0)`**
 - **Usually better to error on the side of too much bias**
 - **adjust to suit the shadow issues in your scene**
 - **Depends somewhat on shadow map precision**
 - **more precision requires less of a bias**
 - **When the shadow map is being magnified, a larger scale is often required**



Render Scene and Access the Depth Texture

- Realizing the theory in practice
 - Fragment's light position can be generated using eye-linear texture coordinate generation
 - specifically OpenGL's `GL_EYE_LINEAR` texgen
 - generate homogenous (s, t, r, q) texture coordinates as light-space (x, y, z, w)
 - T&L engines such as GeForce accelerate texgen!
 - relies on projective texturing

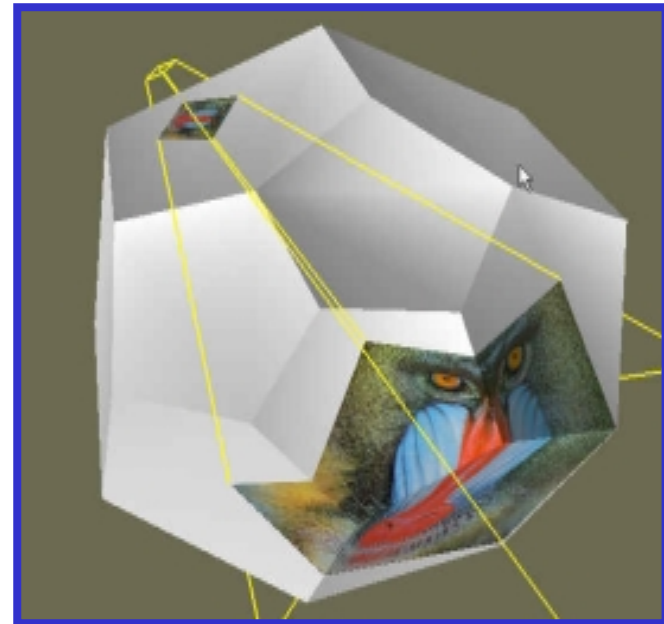


What is Projective Texturing?

- An intuition for projective texturing
 - The slide projector analogy



Source: Wolfgang Heidrich [99]



nVIDIA

About Projective Texturing (1)

- First, what is perspective-correct texturing?
 - Normal 2D texture mapping uses (s, t) coordinates
 - 2D perspective-correct texture mapping
 - means (s, t) should be interpolated linearly in eye-space
 - so compute per-vertex s/w , t/w , and $1/w$
 - linearly interpolated these three parameters over polygon
 - per-fragment compute $s' = (s/w) / (1/w)$ and $t' = (t/w) / (1/w)$
 - results in per-fragment perspective correct (s', t')



About Projective Texturing (2)

- So what is projective texturing?
 - Now consider homogeneous texture coordinates
 - $(s, t, r, q) \rightarrow (s/q, t/q, r/q)$
 - Similar to homogeneous clip coordinates where $(x, y, z, w) = (x/w, y/w, z/w)$
 - Idea is to have $(s/q, t/q, r/q)$ be projected per-fragment
 - This requires a per-fragment divider
 - yikes, dividers in hardware are fairly expensive



About Projective Texturing (3)

- **Hardware designer's view of texturing**
 - **Perspective-correct texturing is a practical requirement**
 - otherwise, textures “swim”
 - perspective-correct texturing already requires the hardware expense of a per-fragment divider
 - **Clever idea [Segal, et.al. '92]**
 - interpolate q/w instead of simply $1/w$
 - so projective texturing is practically free if you already do perspective-correct texturing!



About Projective Texturing (4)

- **Tricking hardware into doing projective textures**
 - By interpolating q/w , hardware computes per-fragment
 - $(s/w) / (q/w) = s/q$
 - $(t/w) / (q/w) = t/q$
 - **Net result: projective texturing**
 - OpenGL specifies projective texturing
 - only overhead is multiplying $1/w$ by q
 - but this is per-vertex



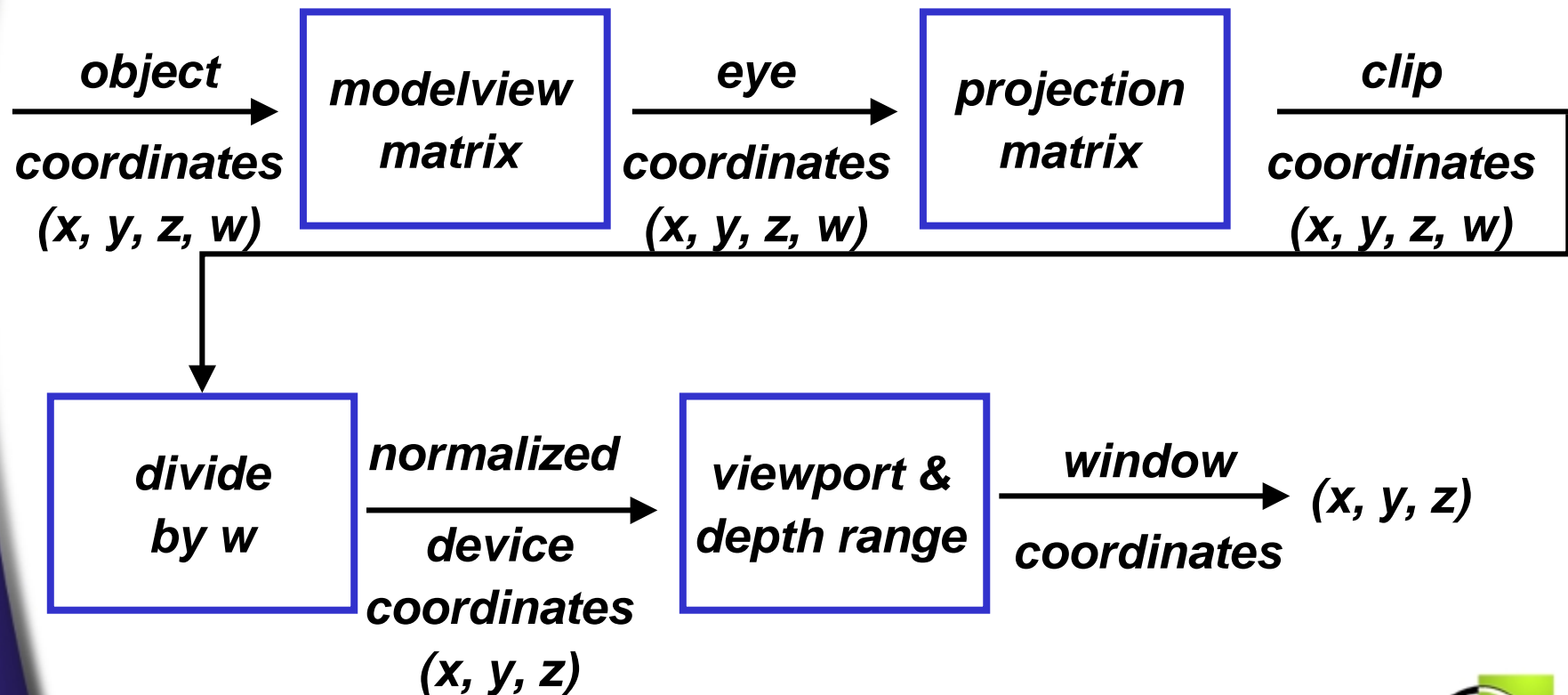
Back to the Shadow Mapping Discussion . . .

- Assign light-space texture coordinates via texgen
 - Transform eye-space (x, y, z, w) coordinates to the light's view frustum (match how the light's depth map is generated)
 - Further transform these coordinates to map directly into the light view's depth map
 - Expressible as a projective transform
 - load this transform into the 4 eye linear plane equations for S, T, and Q coordinates
 - $(s/q, t/q)$ will map to light's depth map texture



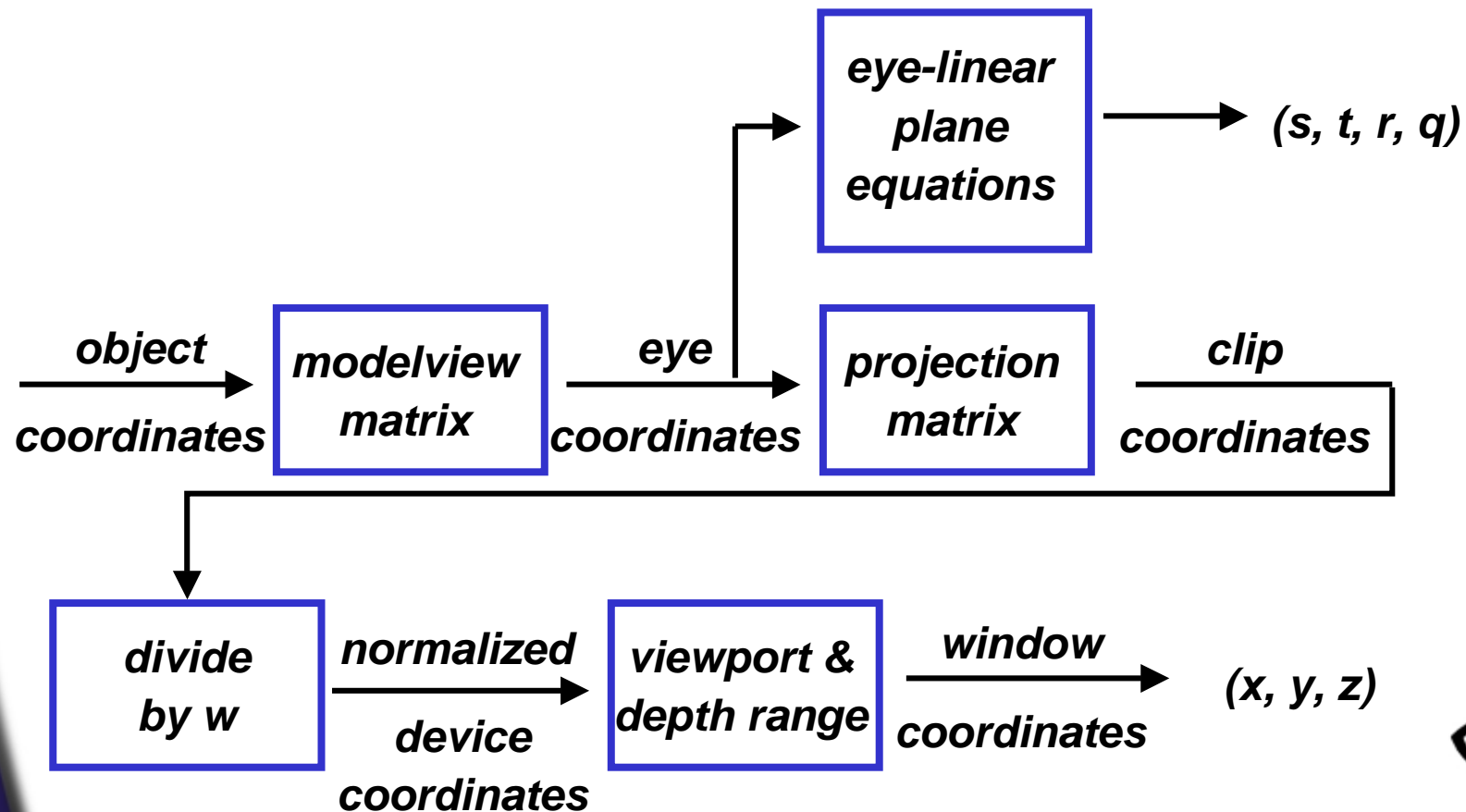
OpenGL's Standard Vertex Coordinate Transform

- From object coordinates to window coordinates



Eye Linear Texture Coordinate Generation

- Generating texture coordinates from eye-space



Setting Up Eye Linear Texgen

- **With OpenGL**

```
GLfloat Splane[4], Tplane[4], Rplane[4], Qplane[4];  
glTexGenfv(GL_S, GL_EYE_PLANE, Splane);  
glTexGenfv(GL_T, GL_EYE_PLANE, Tplane);  
glTexGenfv(GL_R, GL_EYE_PLANE, Rplane);  
glTexGenfv(GL_Q, GL_EYE_PLANE, Qplane);  
glEnable(GL_TEXTURE_GEN_S);  
glEnable(GL_TEXTURE_GEN_T);  
glEnable(GL_TEXTURE_GEN_R);  
glEnable(GL_TEXTURE_GEN_Q);
```

- Each eye plane equation is transformed by current inverse modelview matrix (a very handy thing for us)
 - Note: texgen object planes are *not* transformed by the inverse modelview



Eye Linear Texgen Transform

- Plane equations form a projective transform

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \begin{bmatrix} Splane[0] & Splane[1] & Splane[2] & Splane[3] \\ Tplane[0] & Tplane[1] & Tplane[2] & Tplane[3] \\ Rplane[0] & Rplane[1] & Rplane[2] & Rplane[3] \\ Qplane[0] & Qplane[1] & Qplane[2] & Qplane[3] \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix}$$

- The 4 eye linear plane equations form a 4x4 matrix
(No need for the texture matrix!)



Shadow Map Eye Linear Texgen Transform

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix} = \begin{bmatrix} \text{Eye view} \\ \text{(look at)} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} \text{Modeling} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ w_o \end{bmatrix}$$

glTexGen automatically applies this when modelview matrix contains just the eye view transform

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \begin{bmatrix} 1/2 & 1/2 \\ & 1/2 & 1/2 \\ & & 1/2 & 1/2 \\ & & & 1 \end{bmatrix} \begin{bmatrix} \text{Light frustum} \\ \text{(projection)} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} \text{Light view} \\ \text{(look at)} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} \text{Inverse eye view} \\ \text{(look at)} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix}$$

Supply this combined transform to glTexGen



NVIDIA

Shadow Map Operation

- Automatic depth map lookups
 - After the eye linear texgen with the proper transform loaded
 - $(s/q, t/q)$ is the fragment's corresponding location within the light's depth texture
 - r/q is the Z planar distance of the fragment relative to the light's frustum, scaled and biased to $[0,1]$ range
 - Next compare texture value at $(s/q, t/q)$ to value r/q
 - if $\text{texture}[s/q, t/q] \cong r/q$ then *not shadowed*
 - if $\text{texture}[s/q, t/q] < r/q$ then *shadowed*



Dedicated Hardware Shadow Mapping Support

- **SGL RealityEngine, InfiniteReality, GeForce3 and Xbox Hardware**
 - Performs the shadow test as a texture filtering operation
 - looks up texel at $(s/q, t/q)$ in a 2D texture
 - compares lookup value to r/q
 - if texel is greater than or equal to r/q , then generate 1.0
 - if texel is less than r/q , then generate 0.0
 - Modulate color with result
 - zero if fragment is shadowed or unchanged color if not



OpenGL Extensions for Shadow Map Hardware

- **Two extensions work together**
 - **SGIX_depth_texture**
 - supports high-precision depth texture formats
 - copy from depth buffer to texture memory supported
 - **SGIX_shadow**
 - adds “shadow comparison” texture filtering mode
 - compares r/q to texel value at $(s/q, t/q)$
- **Multi-vendor support: SGI, NVIDIA, others?**
 - Brian Paul has implemented these extensions in Mesa!



New Depth Texture Internal Texture Formats

- **SGIX_depth_texture** supports textures containing depth values for shadow mapping
- Three new internal formats
 - **GL_DEPTH_COMPONENT16_SGIX**
 - **GL_DEPTH_COMPONENT24_SGIX**
 - **GL_DEPTH_COMPONENT32_SGIX**
(same as 24-bit on GeForce3)
- Use **GL_DEPTH_COMPONENT** for your external format
- Work with **glCopySubTexImage2D** for fast copies from depth buffer to texture
 - **NVIDIA** optimizes these copy texture paths



Depth Texture Details

- **Usage example:**

```
glCopyTexImage2D(GL_TEXTURE_2D, level=0,  
    internalfmt=GL_DEPTH_COMPONENT24_SGIX,  
    x=0, y=0, w=256, h=256, border=0);
```
- **Then use glCopySubTexImage2D for faster updates once texture internal format initially defined**
- **Hint: use GL_DEPTH_COMPONENT for your texture internal format**
 - **Leaving off the “n_SGIX” precision specifier tells the driver to match your depth buffer’s precision**
 - **Copy texture performance is optimum when depth buffer precision matches the depth texture precision**



Depth Texture Copy Performance

- The more depth values you copy, the slower the performance
 - 512x512 takes 4 times longer to copy than 256x256
 - Tradeoff: better defined shadows require higher resolution shadow maps, but slows copying
- 16-bit depth values copy twice as fast as 24-bit depth values (which are contained in 32-bit words)
 - Requesting a 16-bit depth buffer (even with 32-bit color buffer) and copying to a 16-bit depth texture is faster than using a 24-bit depth buffer
 - Note that using 16-bit depth buffer usually requires giving up stencil



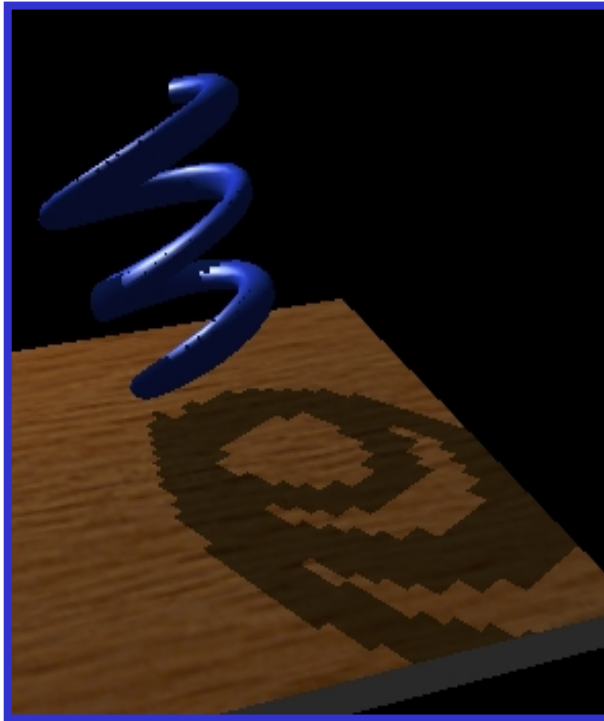
Hardware Shadow Map Filtering

- **“Percentage Closer” filtering**
 - Normal texture filtering just averages color components
 - Averaging depth values does NOT work
 - Solution [Reeves, SIGGRAPH 87]
 - Hardware performs comparison for each sample
 - Then, averages results of comparisons
 - Provides anti-aliasing at shadow map edges
 - Not soft shadows in the umbra/penumbra sense

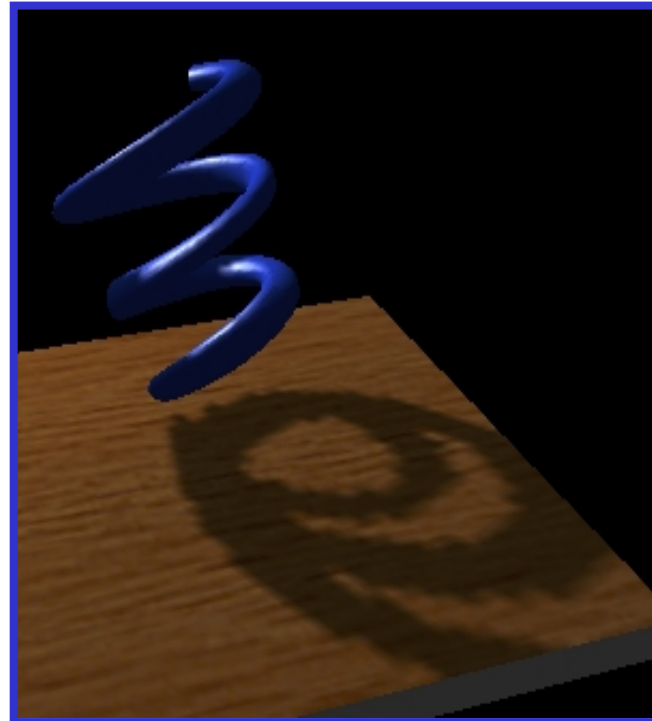


Hardware Shadow Map Filtering Example

GL_NEAREST: blocky



GL_LINEAR: antialiased edges



***Low shadow map resolution
used to heightens filtering artifacts***

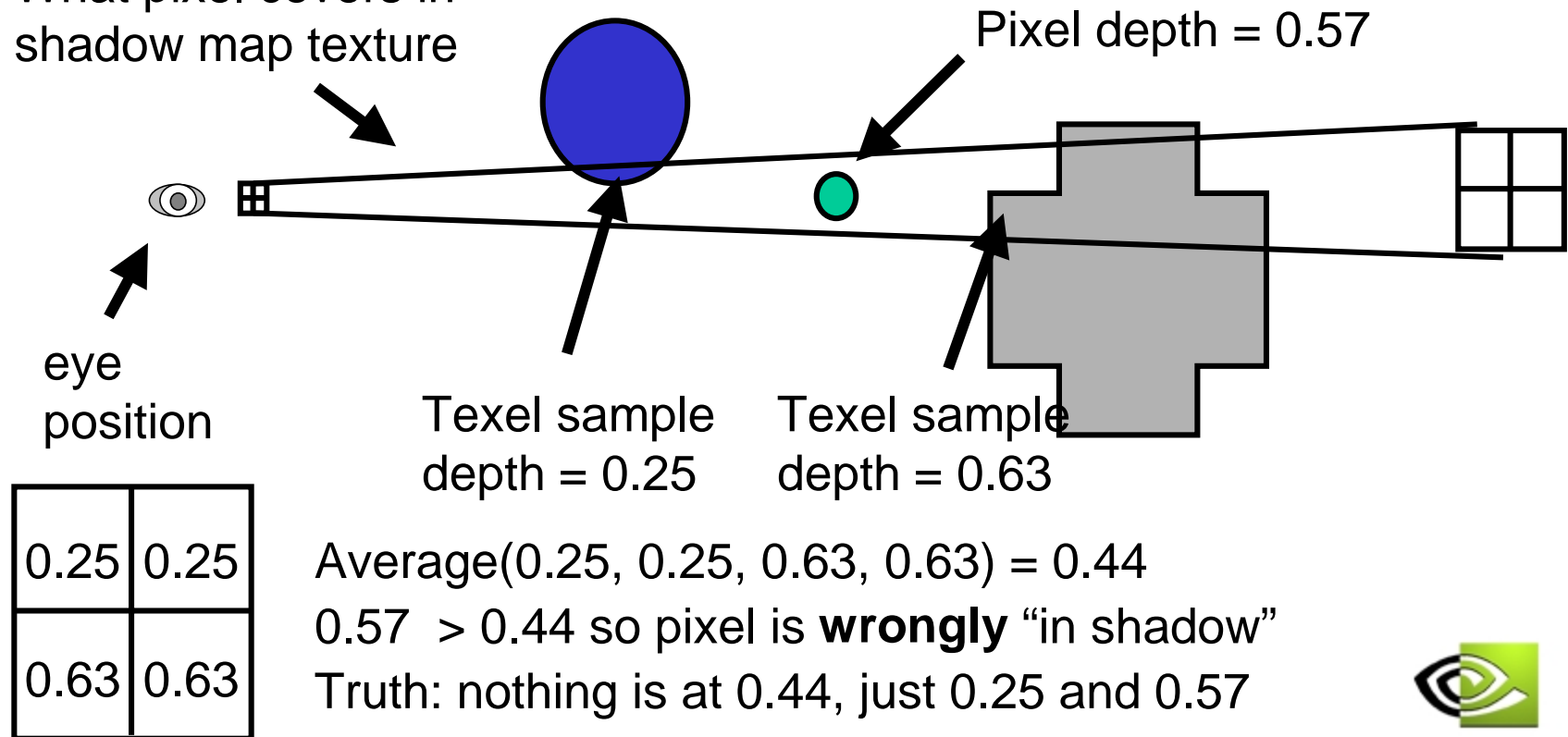


NVIDIA

Depth Values are not Blend-able

- Traditional filtering is inappropriate

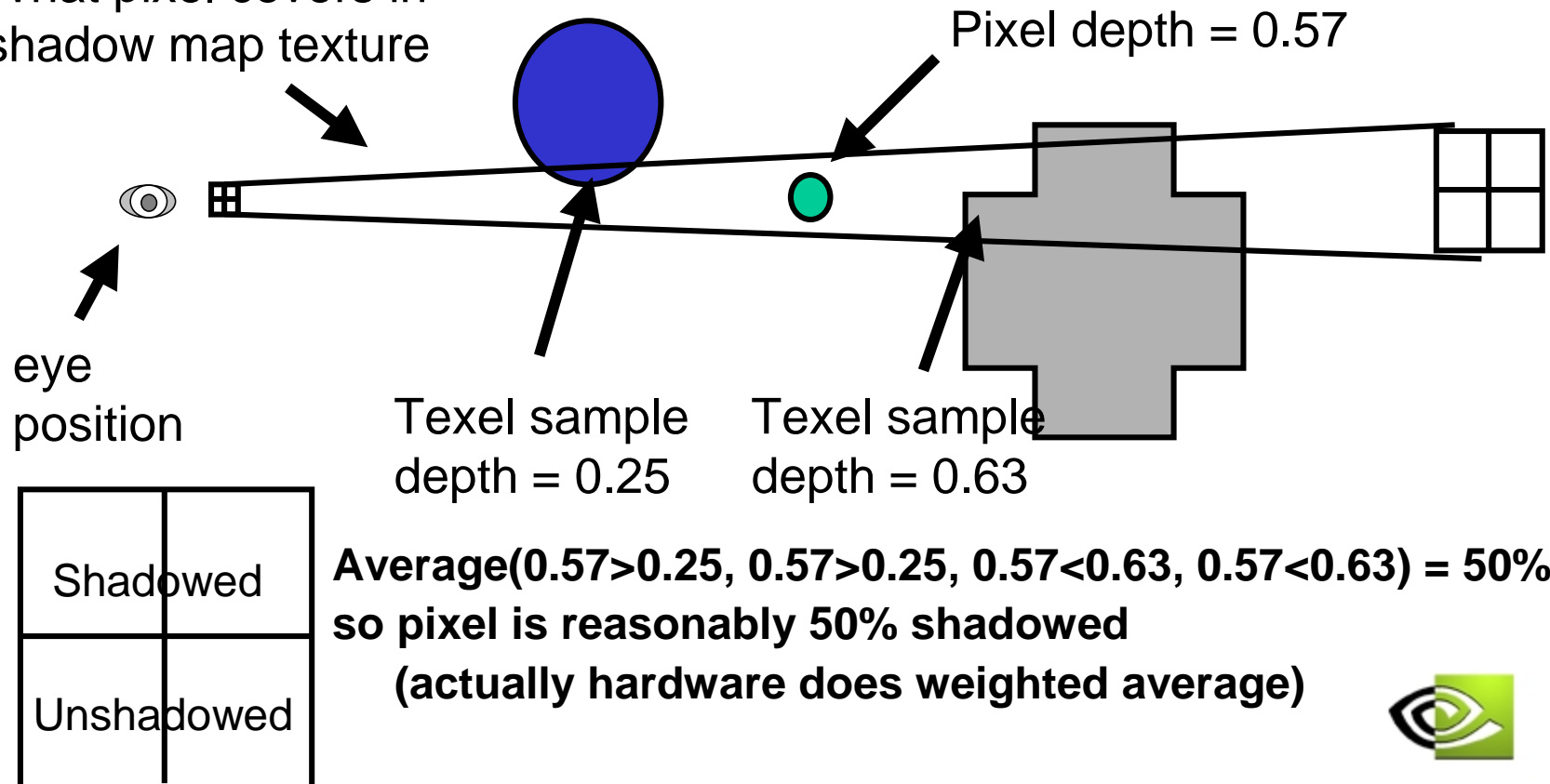
What pixel covers in
shadow map texture



Percentage Closer Filtering

- Average comparison *results*, not depth values**

What pixel covers in
shadow map texture



Average(0.57 > 0.25, 0.57 > 0.25, 0.57 < 0.63, 0.57 < 0.63) = 50%
so pixel is reasonably 50% shadowed
(actually hardware does weighted average)



NVIDIA

Mipmap Filtering for Depth Textures with Percentage Closer Filtering (1)

- Mipmap filtering works
 - Averages the results of comparisons from the one or two mipmap levels sampled
- You *cannot* use `gluBuild2DMipmaps` to construct depth texture mipmaps
 - Again, because you cannot blend depth values!
- If you do want mipmaps, the best approach is re-rendering the scene at each required resolution
 - Usually too expensive to be practical for all mipmap levels
 - OpenGL 1.2 LOD clamping can help avoid rendering all the way down to the 1x1 level



Mipmap Filtering for Depth Textures with Percentage Closer Filtering (2)

- Mipmaps can make it harder to find an appropriate polygon offset scale & bias that guarantee avoidance of self-shadowing
- You can get “8-tap” filtering by using (for example) two mipmap levels, 512x512 and 256x256, and setting your min and max LOD clamp to 0.5
 - Uses OpenGL 1.2 LOD clamping



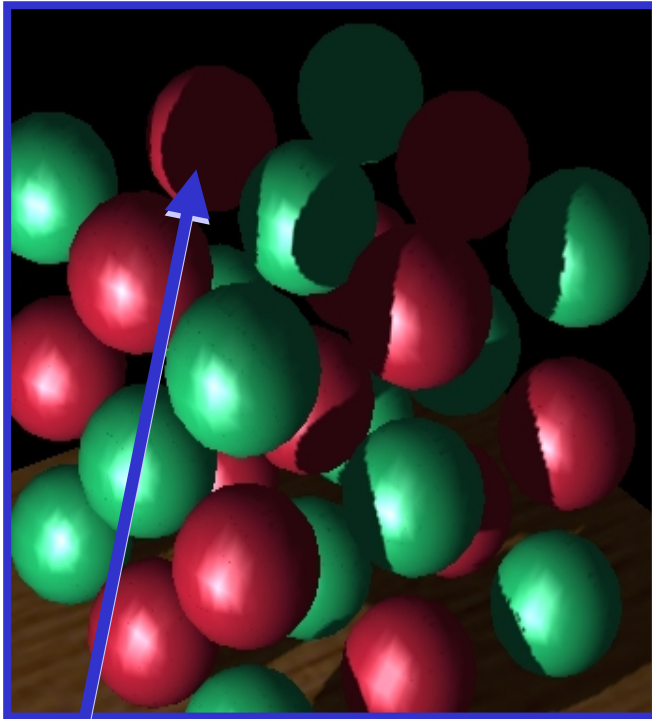
Advice for Shadowed Illumination Model (1)

- **Typical illumination model with decal texture:**
 $(ambient + diffuse) * decal + specular$
- **The shadow map supplies a shadowing term**
- **Assume shadow map supplies a shadowing term, *shade***
 - **Percentage shadowed**
 - **100% = fully visible, 0% = fully shadowed**
- **Obvious updated illumination model for shadowing:**
 $(ambient + shade * diffuse) * decal + shade * specular$
- **Problem is real-world lights don't 100% block diffuse shading on shadowed surfaces**
 - **Light scatters; real-world lights are not ideal points**



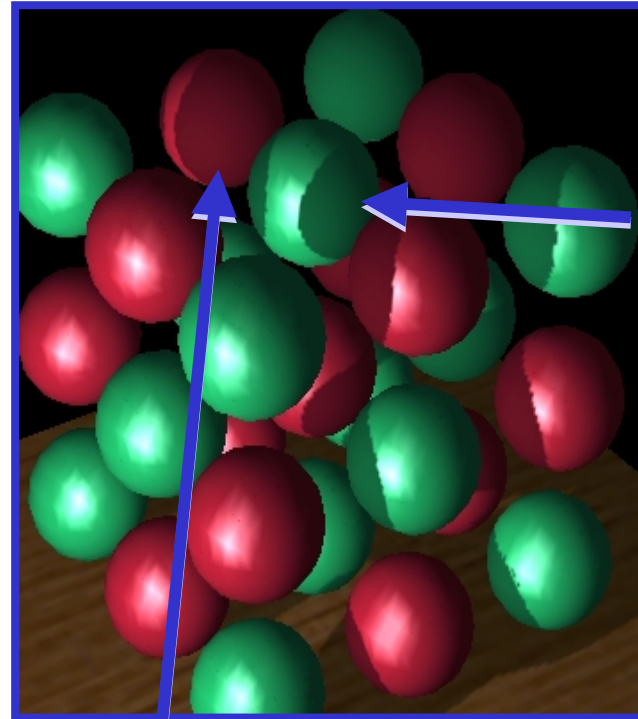
The Need for Dimming Diffuse

No dimming; shadowed regions have 0% diffuse and 0% specular



Front facing shadowed regions appear unnaturally flat.

With dimming; shadowed regions have 40% diffuse and 0% specular



Still evidence of curvature in shadowed regions.

No specular in shadowed regions in both versions.



nVIDIA

Advice for Shadowed Illumination Model (2)

- **Illumination model with dimming:**

*(ambient + diffuseShade * diffuse) * decal + specular * shade*

where *diffuseShade* is

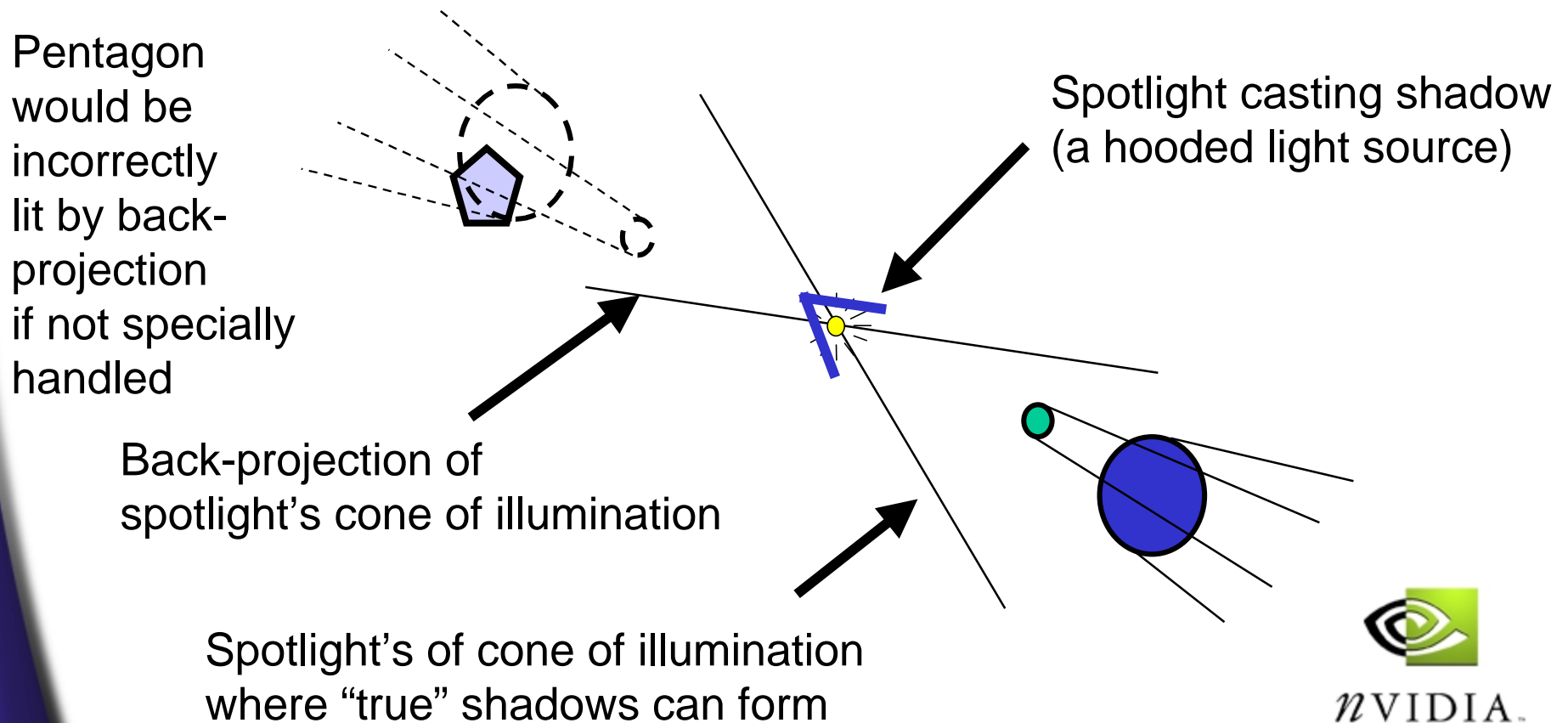
*diffuseShade = dimming + (1.0 - dimming) * shade*

- **Easy to implement with NV_register_combiners & OpenGL 1.2 “separate specular color” support**
 - Separate specular keeps the diffuse & specular per-vertex lighting results distinct
 - NV_register_combiners can combine the primary (diffuse) and secondary (specular) colors per-pixel with the above math



Careful about Back Projecting Shadow Maps (1)

- Just like standard projective textures, shadow maps can back-project



Careful about Back Projecting Shadow Maps (2)

- **Techniques to eliminate back-projection:**
 - Modulate shadow map result with lighting result from a single per-vertex spotlight with the proper cut off (ensures is light “off” behind the spotlight)
 - Use a small 1D texture where “s” is planar distance from the light (generated “s” with a planar texgen mode), then 1D texture is 0.0 for negative distances and 1.0 for positive distances.
 - Use a clip plane positioned at the plane defined by the light position and spotlight direction
 - Simply avoid drawing geometry “behind” the light when applying the shadow map (better than a clip plane)
 - NV_texture_shader’s GL_PASS_THROUGH_NV mode



Other Useful OpenGL Extensions for Improving Shadow Mapping

- **ARB_pbuffer** – create off-screen rendering surfaces for rendering shadow map depth buffers
 - Normally, you can construct shadow maps in your back buffer and copy them to texture
 - But if the shadow map resolution is larger than your window resolution, use pbuffers.
- **NV_texture_rectangle** – new 2D texture target that does not require texture width and height to be powers of two
 - Limitations
 - No mipmaps or mipmap filtering supported
 - No wrap clamp mode
 - Texture coords in $[0..w] \times [0..h]$ rather than $[0..1] \times [0..1]$ range.
 - Quite acceptable for for shadow mapping



Combining Shadow Mapping with other Techniques

- **Good in combination with techniques**
 - Use stencil to tag pixels as inside or outside of shadow
 - Use other rendering techniques in extra passes
 - bump mapping
 - texture decals, etc.
 - Shadow mapping can be integrated into more complex multi-pass rendering algorithms
- **Shadow mapping algorithm does not require access to vertex-level data**
 - Easy to mix with vertex programs and such



An Alternative to Dedicated Shadow Mapping Hardware

- **Consumer 3D hardware solution**
 - Proposed by Wolfgang Heidrich in his 1999 Ph.D. thesis
 - Leverages today's consumer multi-texture hardware
 - 1st texture unit accesses 2D depth map texture
 - 2nd texture unit accesses 1D Z range texture
 - Extended texture environment subtracts 2nd texture from 1st
 - shadowed if greater than zero, unshadowed otherwise
 - use alpha test to discard shadowed fragments



Issues with Shadow Mapping (1)

- **Not without its problems**
 - **Prone to aliasing artifacts**
 - “percentage closer” filtering helps this
 - normal color filtering does not work well
 - **Depth bias is not completely foolproof**
 - **Requires extra shadow map rendering pass and texture loading**
 - **Higher resolution shadow map reduces blockiness**
 - but also increase texture copying expense



Issues with Shadow Mapping (2)

- **Not without its problems**
 - **Shadows are limited to view frustums**
 - **could use six view frustums for omni-directional light**
 - **Objects outside or crossing the near and far clip planes are not properly accounted for by shadowing**
 - **move near plane in as close as possible**
 - **but too close throws away valuable depth map precision when using a projective frustum**



Some Theory for Determining Your Shadow Map Resolution (1)

- Requires knowing how pixels (samples) in the light's view compare to the size of pixels (samples) in the eye's view
 - A re-sampling problem
- When light source frustum is reasonably well aligned with the eye's view frustum, the ratio of sample sizes is close to 1.0
 - Great match if eye and light frustum's are nearly identical
 - But that implies very few viewable shadows
 - Consider a miner's lamp (i.e., a light attached to your helmet)
 - The chief reason for such a lamp is you don't see shadows from the lamp while wearing it



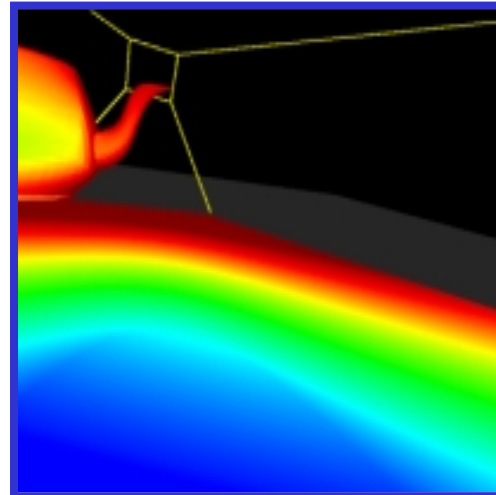
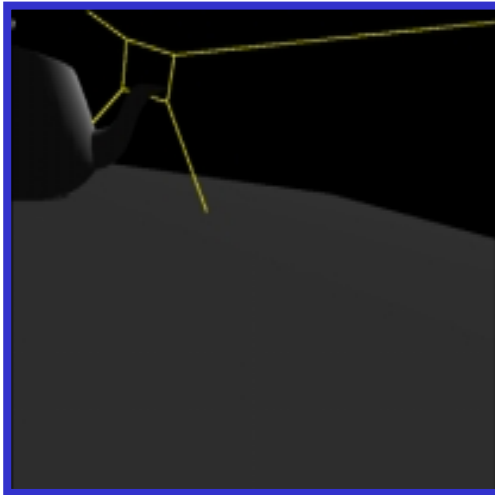
Some Theory for Determining Your Shadow Map Resolution (2)

- So best case is miner's lamp
- Worst case is shadows from light shining at the viewer
 - “that deer in the headlights” problem – definitely worst case for the deer
 - Also known as the “dueling frusta” problem (frusta, plural of frustum)
- Let's attempt to visualize what's happens...



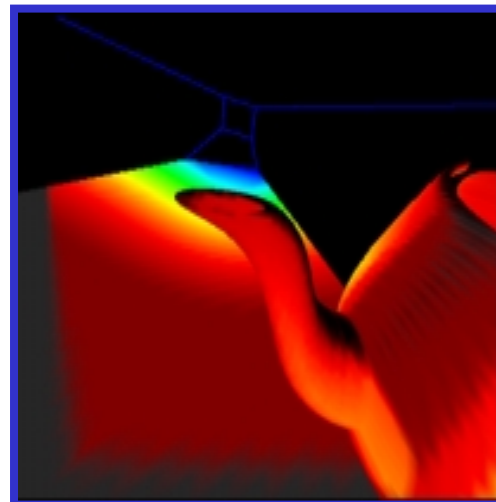
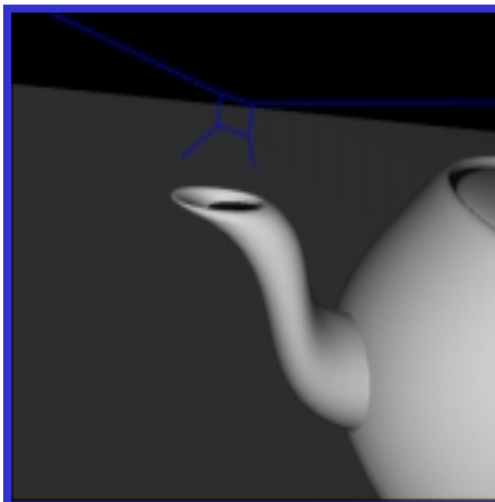
Four Images of Dueling Frusta Case

*Eye's
View*



*Eye's View with
projection
of color-coded
mipmap levels
from light:
Blue =
magnification
Red = minification*

*Light's
View*



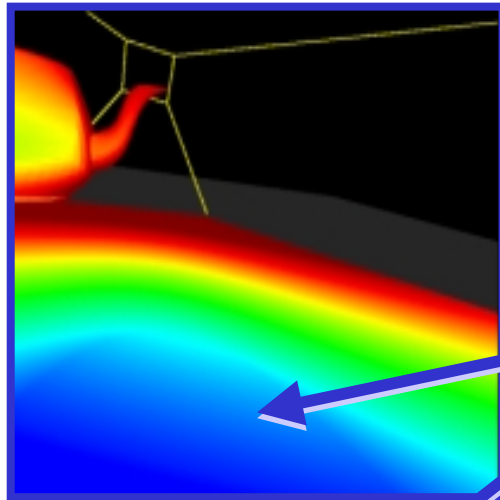
*Light's View with
re-projection
of above image
from the eye*



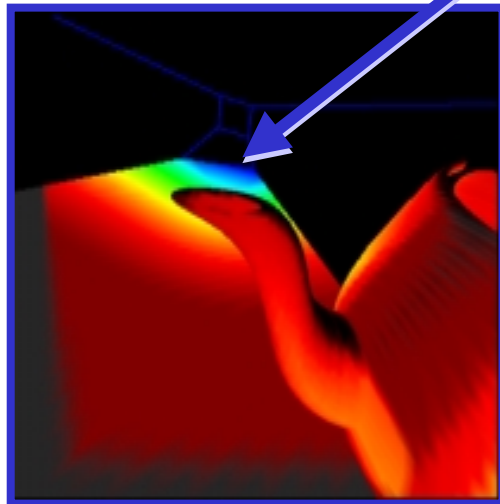
nVidia

Interpretation of the Four Images of the Dueling Frusta Case

Eye's View



Light's View



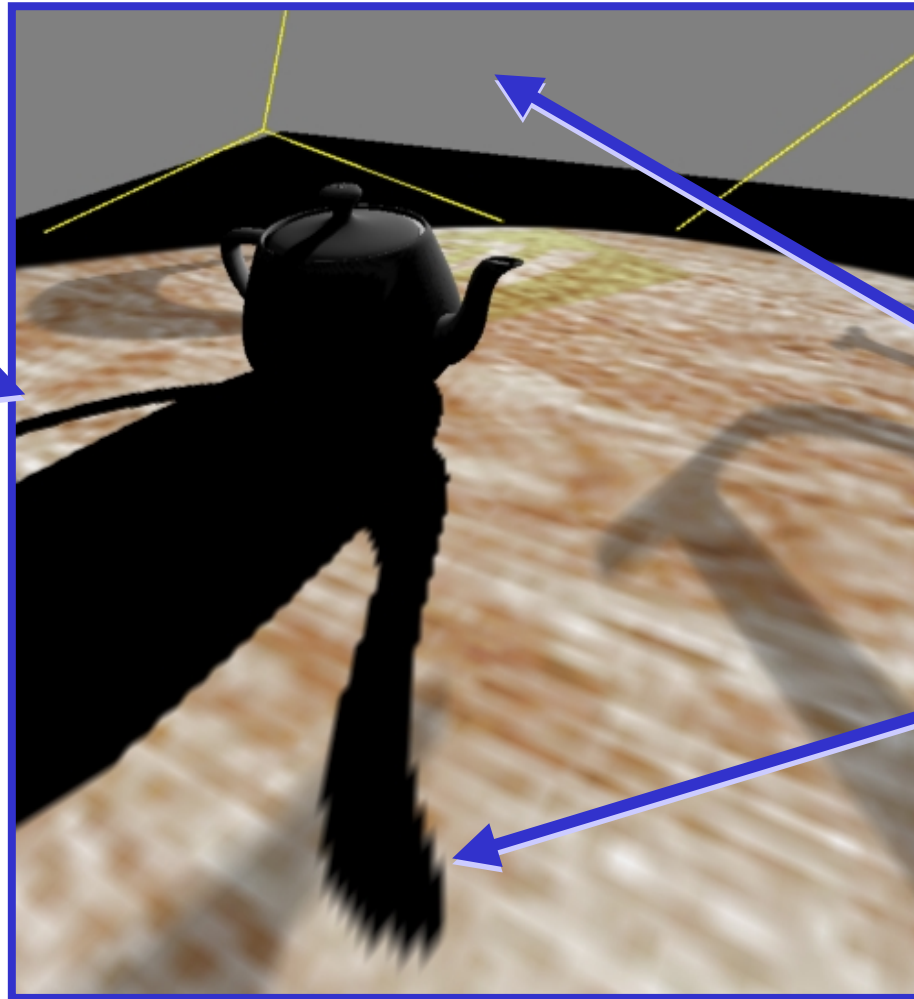
Region that is smallest in the light's view is a region that is very large in the eye's view. This implies that it would require a very high-resolution shadow map to avoid obvious blocky shadow edge artifacts.



NVIDIA™

Example of Blocky Shadow Edge Artifacts in Dueling Frusta Situations

Notice that shadow edge is well defined in the distance.



Light position out here pointing towards the viewer.

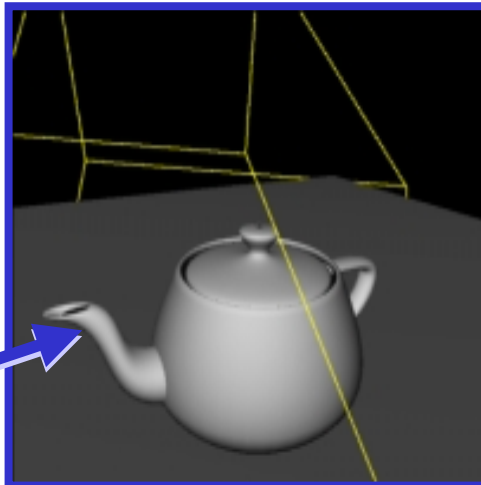
Blocky shadow edge artifacts.



NVIDIA™

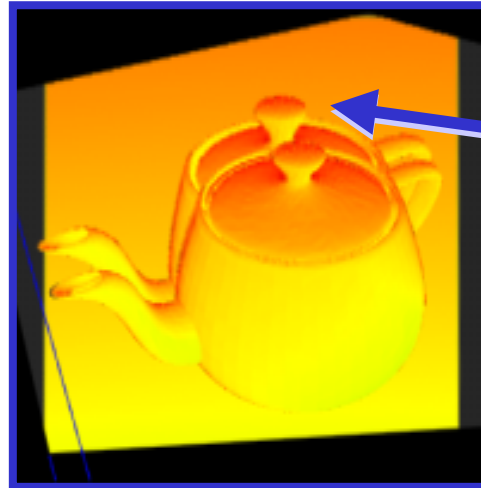
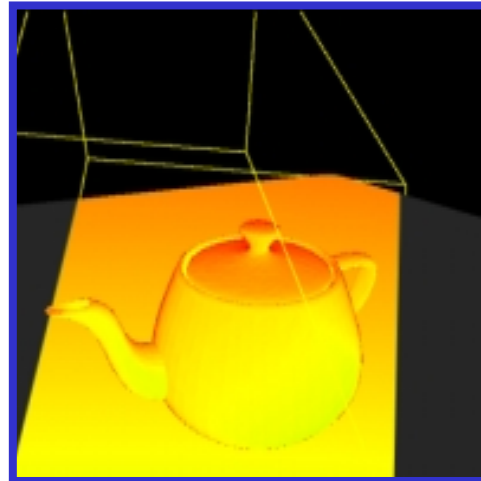
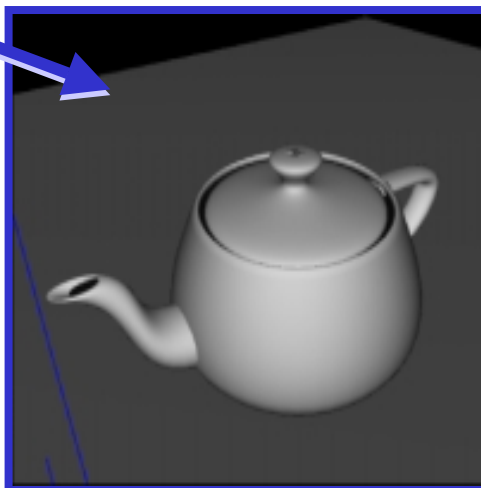
Good Situation, Close to the Miner's Lamp

Eye's View



Very similar views

Light's View



Note how the color-coded images share similar pattern and the coloration is uniform. Implies single depth map resolution would work well for most of the scene.

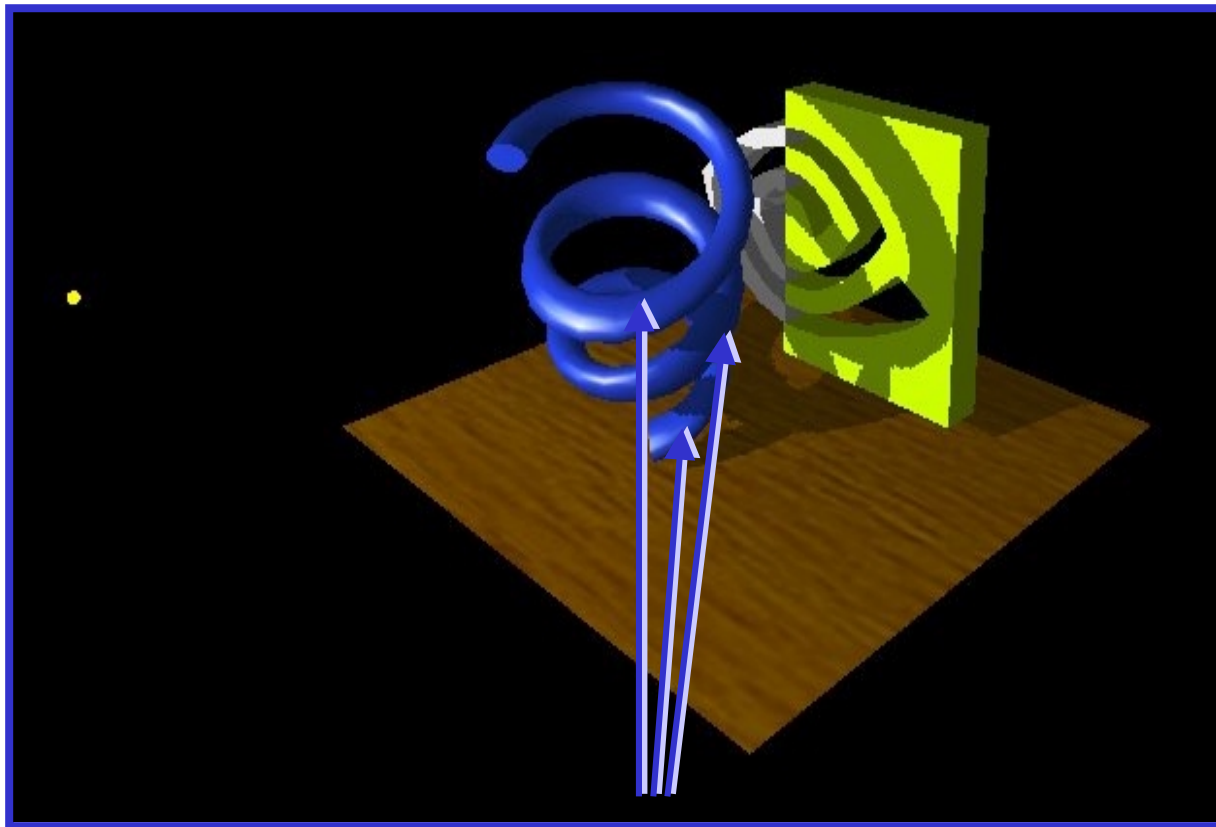
Ghosting is where projection would be in shadow.



nVIDIA™

More Examples

- Smooth surfaces with object self-shadowing



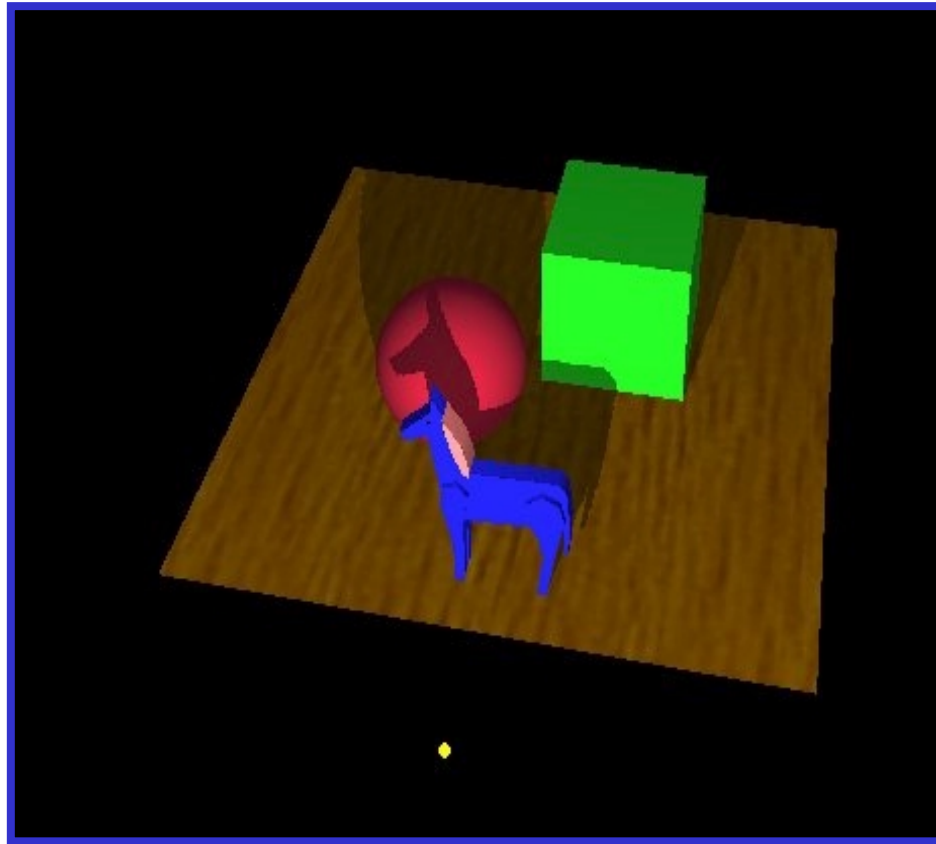
Note object self-shadowing



nVIDIA™

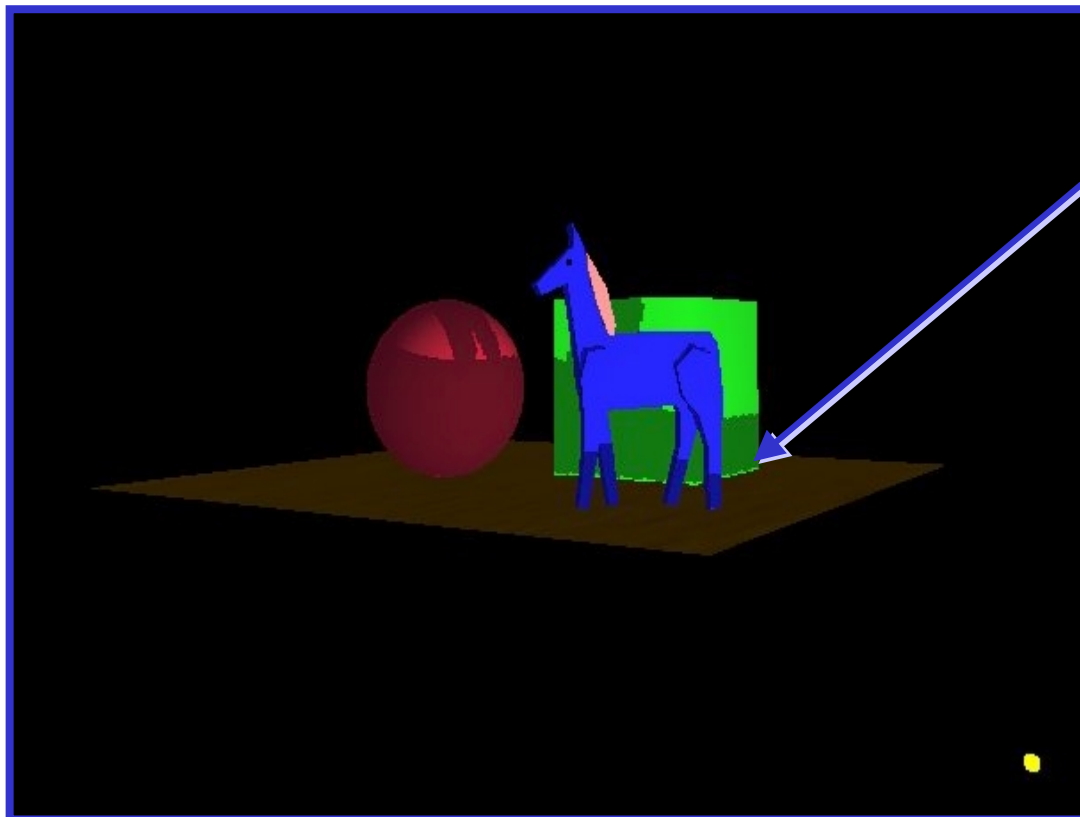
More Examples

- **Complex objects all shadow**



More Examples

- Even the floor casts shadow



Note shadow leakage due to infinitely thin floor

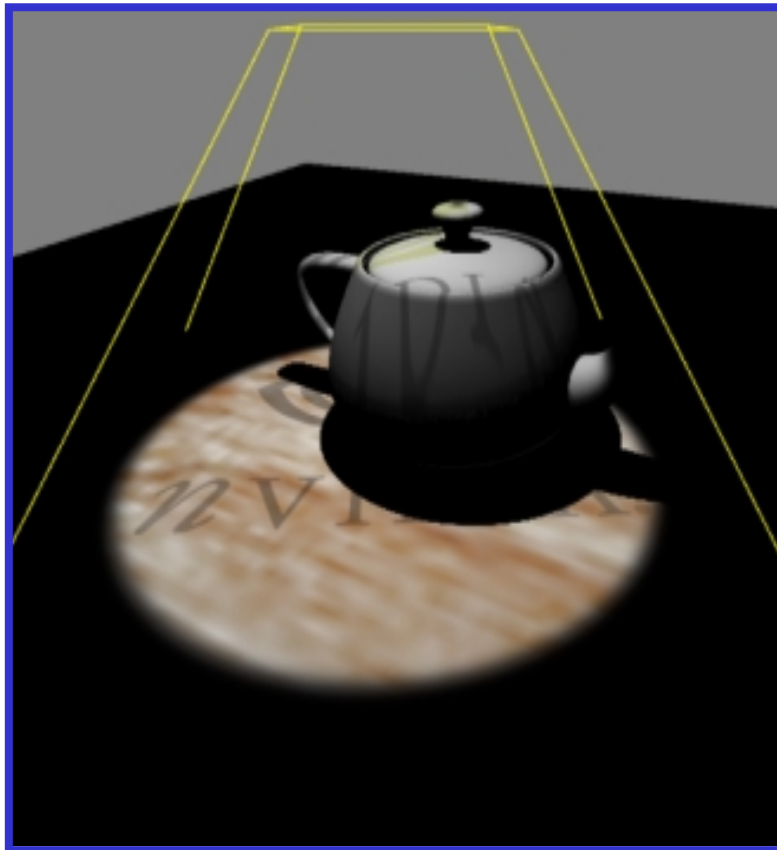
Could be fixed by giving floor thickness



NVIDIA™

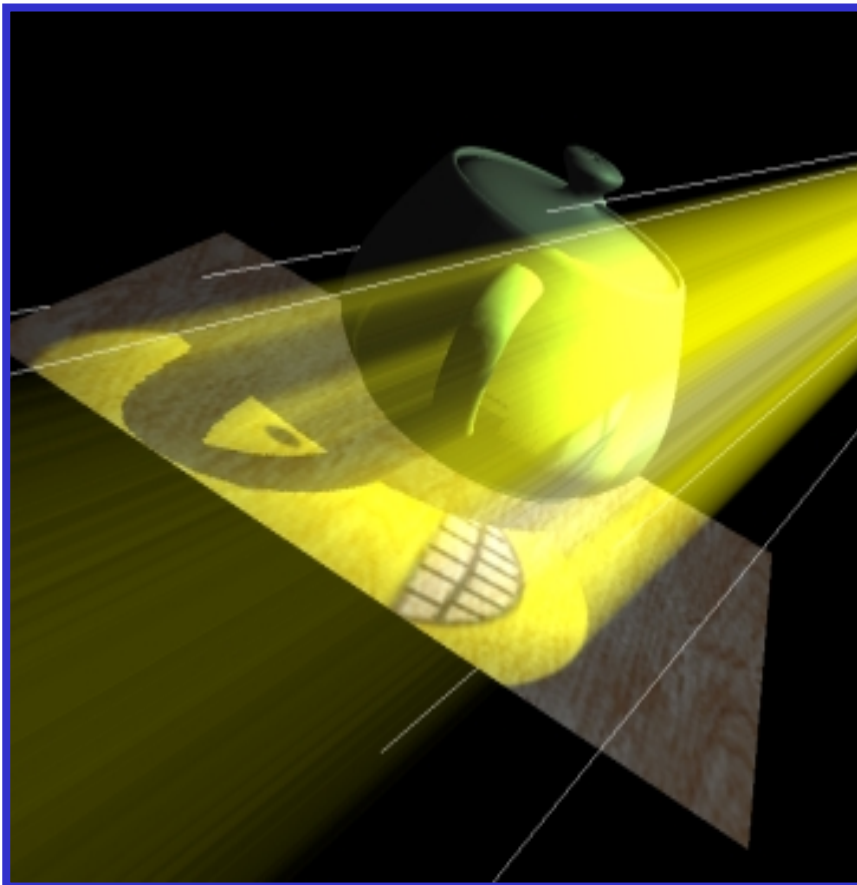
Combine with Projective Texturing for Spotlight Shadows

- Use a spotlight-style projected texture to give shadow maps a spotlight falloff



Combining Shadows with Atmospherics

- Shadows in a dusty room



Simulate atmospheric effects such as suspended dust

- 1) *Construct shadow map*
- 2) *Draw scene with shadow map*
- 3) *Modulate projected texture image with projected shadow map*
- 4) *Blend back-to-front shadowed slicing planes also modulated by projected texture image*

Credit: Cass Everitt



NVIDIA

Luxo Jr. in Real-time using Shadow Mapping

- Steve Jobs at MacWorld Japan shows this on a Mac with OpenGL using hardware shadow mapping



nVIDIA™

Luxo Jr. Demo Details

- Luxo Jr. has two animated lights and one overhead light
 - Three shadow maps *dynamically* generated per frame
- Complex geometry (cords and lamp arms) all correctly shadowed
- User controls the view, shadowing just works
- Real-time Luxo Jr. is technical triumph for OpenGL
- Only available in OpenGL.



(Sorry, no demo. Images are from web cast video of Apple's MacWorld Japan announcement.)



nVIDIA™

Shadow Mapping Source Code

- Find it on the NVIDIA web site
 - The source code
 - “shadowcast” in OpenGL example code
 - Works on TNT, GeForce, Quadro, & GeForce3 using best available shadow mapping support
 - And vendors that support EXT_texture_env_combine
 - *NVIDIA OpenGL Extension Specifications*
 - documents EXT_texture_env_combine, NV_register_combiners, SGIX_depth_texture, & SGIX_shadow
 - <http://www.nvidia.com>



Credits

- **The inspiration for these ideas**
 - **Wolfgang Heidrich, Max-Planck Institute for Computer Science**
 - original dual-texture shadow mapping idea
 - read his thesis *High-quality Shading and Lighting for Hardware-accelerated Rendering*
 - **Michael McCool, University of Waterloo**
 - suggested idea for multi-digit shadow comparisons



Conclusions

- **Shadow mapping offers real-time shadowing effects**
 - **Independent of scene complexity**
 - **Very compatible with multi-texturing**
 - **Does not mandate multi-pass as stenciled shadow volumes do**
 - **Ideal for shadows from spotlights**
- **Consumer hardware shadow map support here today**
 - **GeForce3**
 - **Dual-texturing technique supports legacy hardware**
- **Same basic technique used by Pixar to generate shadows in their computer-generated movies**

