

***GDC 2000:***  
***Programming Session***

**Shadow Mapping**  
**with Today's OpenGL Hardware**

***March 10, 2000***

**Mark J. Kilgard**

**Graphics Software Engineer**

**NVIDIA Corporation**

# Motivation for Better Shadows

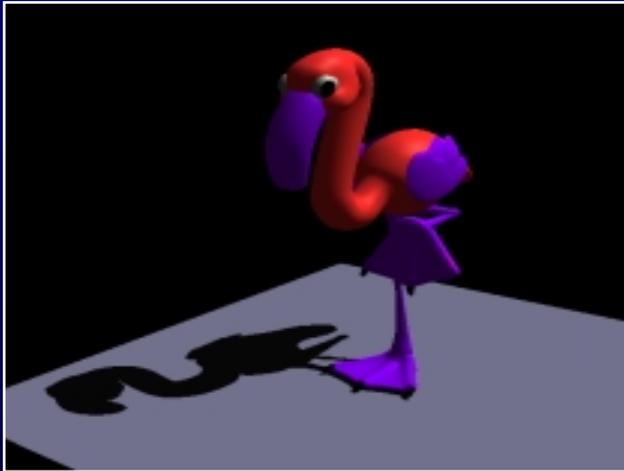
---



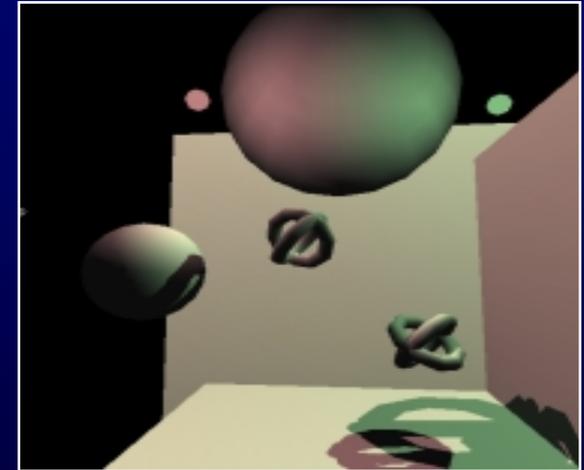
## *Shadows increase scene realism*

- 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



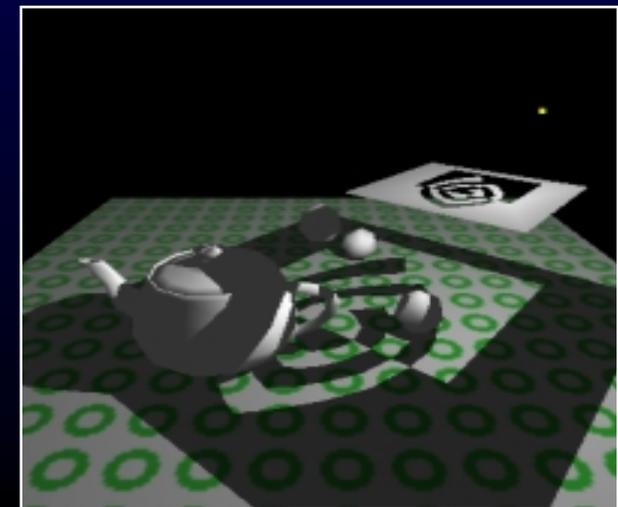
*Projected  
planar  
shadows*



*Shadow  
volumes*



*Light maps*



*Hybrid  
approaches*

# Problems with Common Shadow Techniques



## *Mostly hacks 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

# 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

# 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)

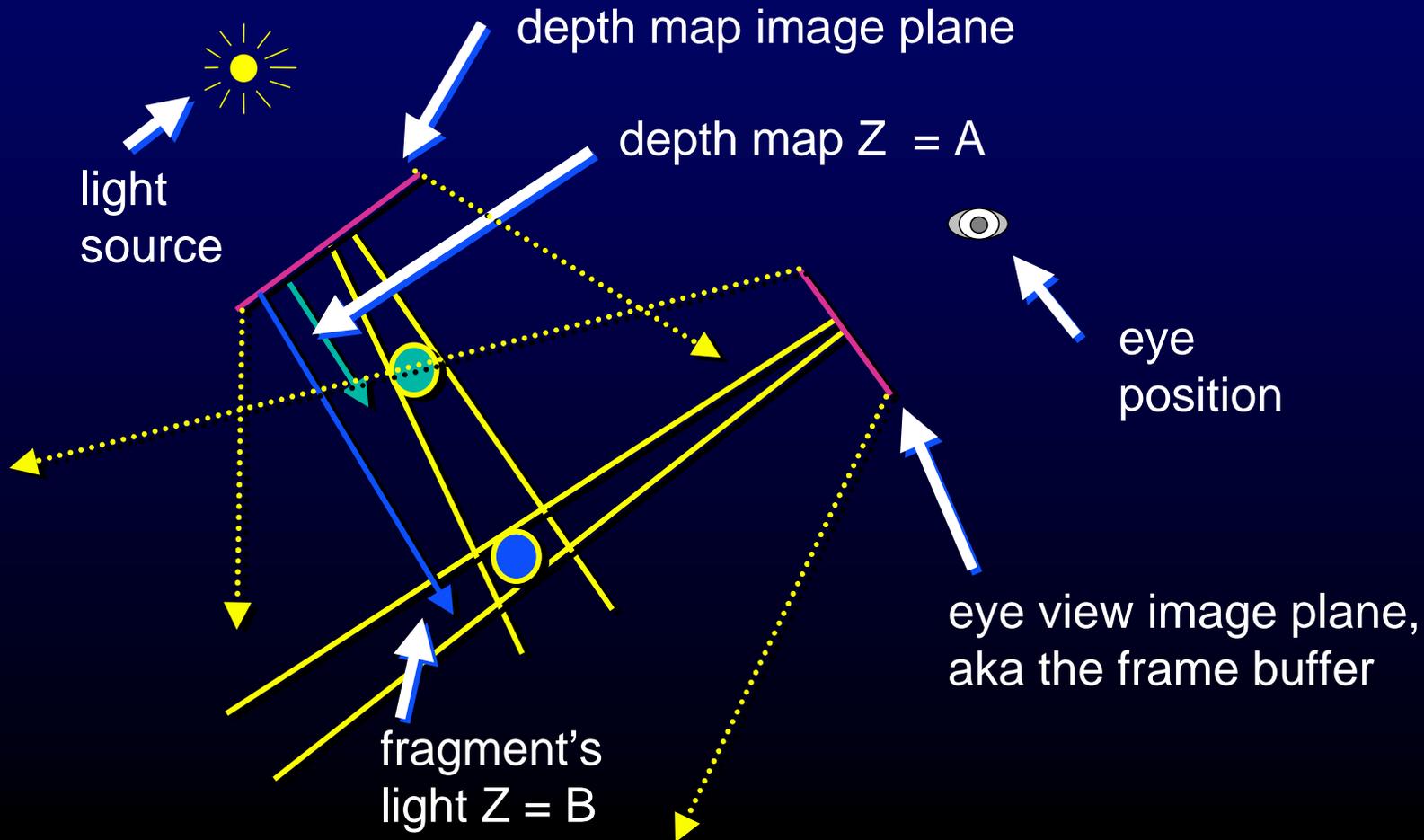


## *The Shadow Map Comparison*

- 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

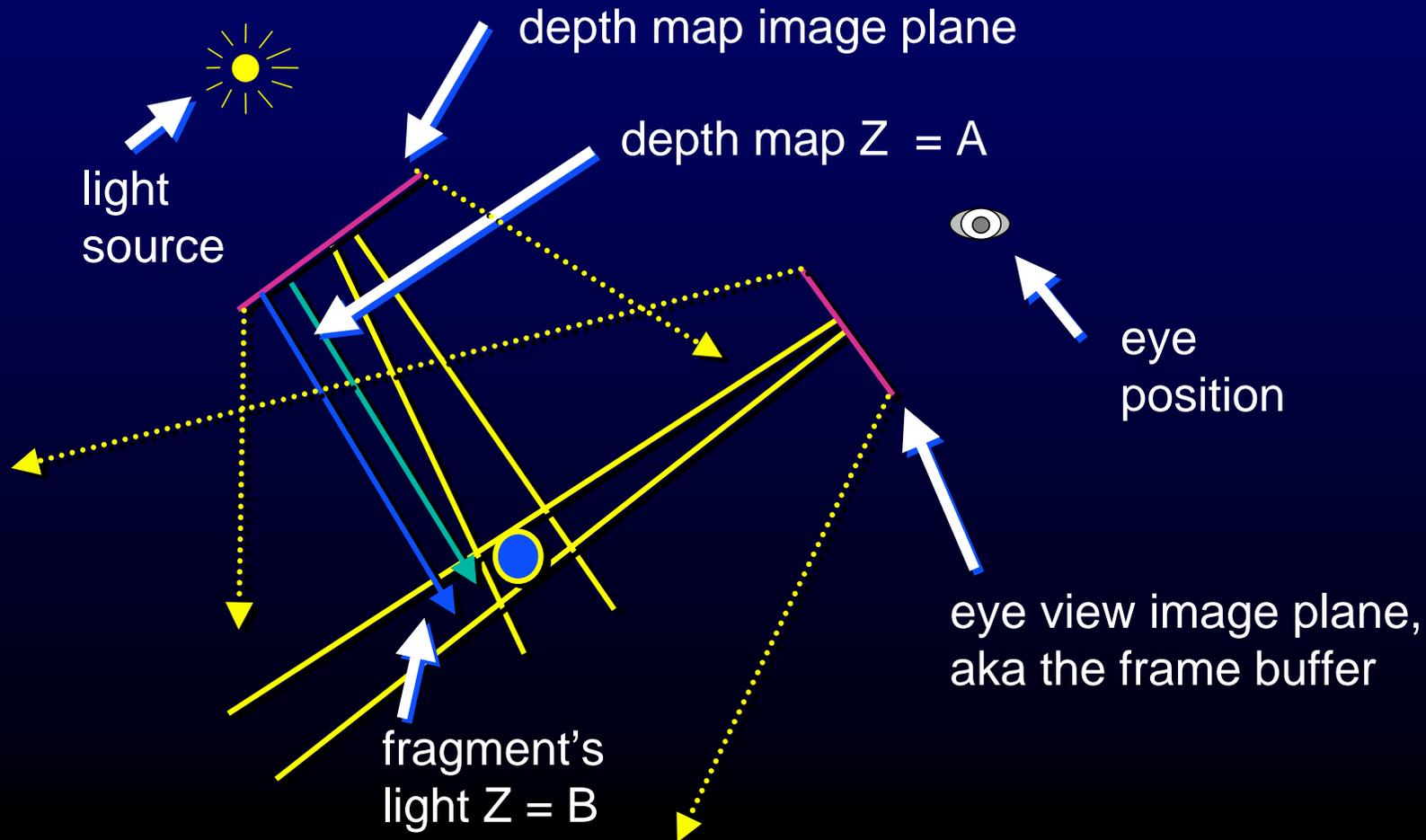
# Shadow Mapping with a picture in 2D

## *The $A < B$ shadowed fragment case*



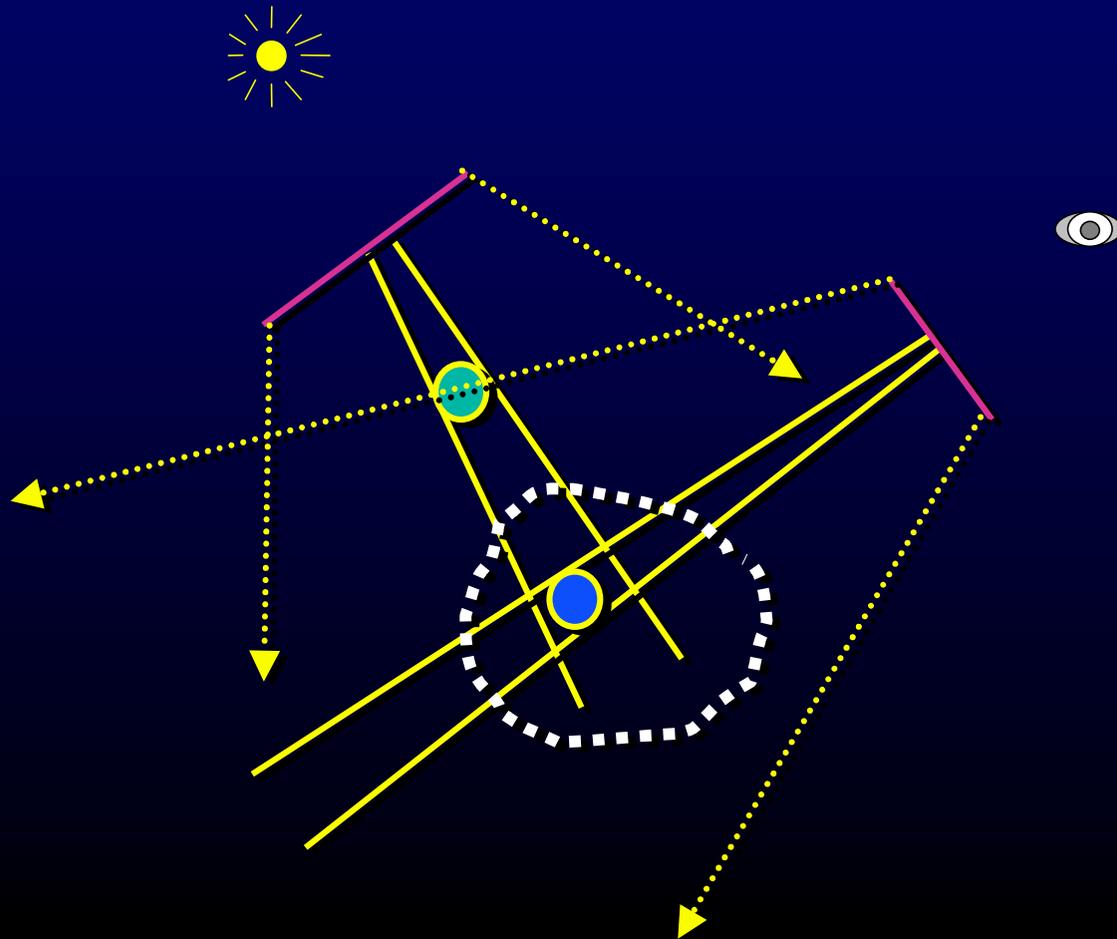
# Shadow Mapping with a picture in 2D

## *The $A \cong B$ unshadowed fragment case*



# Shadow Mapping with a picture in 2D

***Note image precision mismatch!***



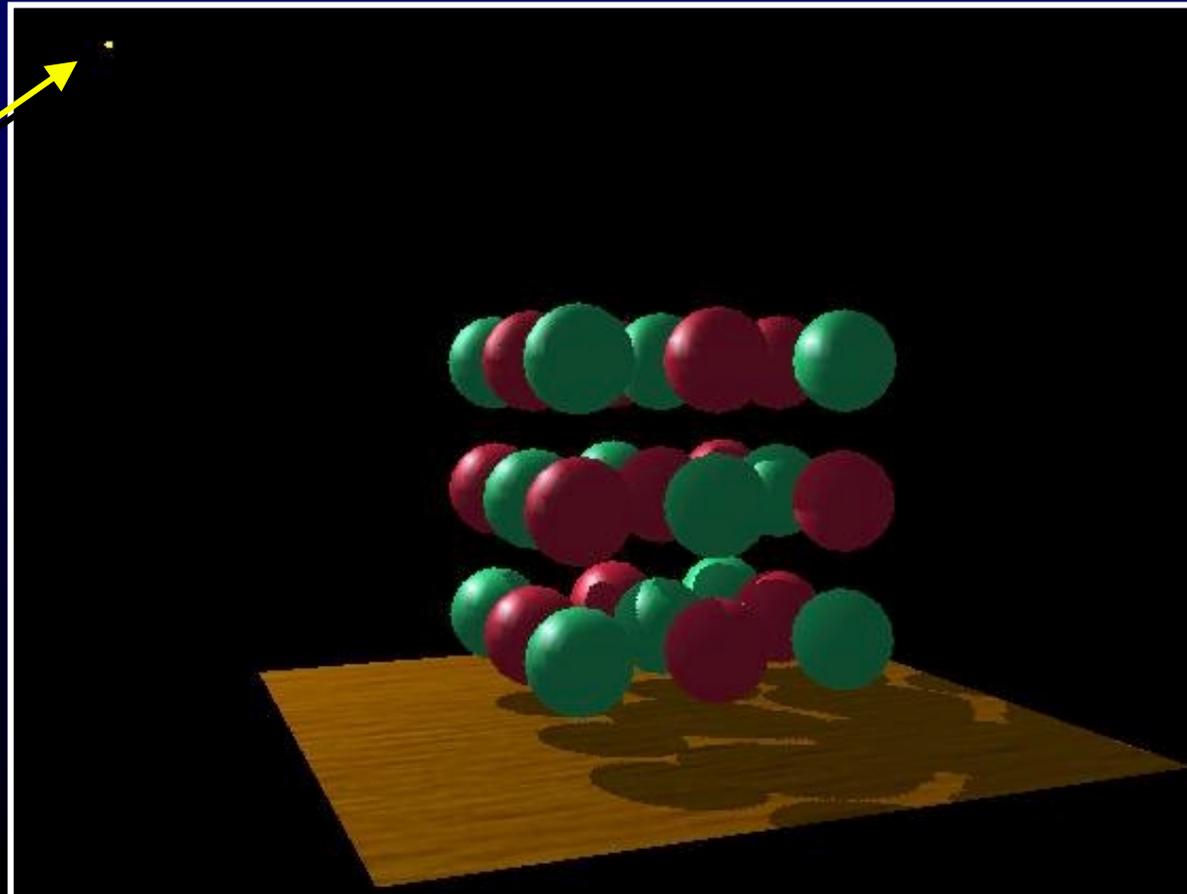
The depth map  
could be at a  
different resolution  
from the framebuffer

This mismatch can  
lead to artifacts

# Visualizing the Shadow Mapping Technique (1)

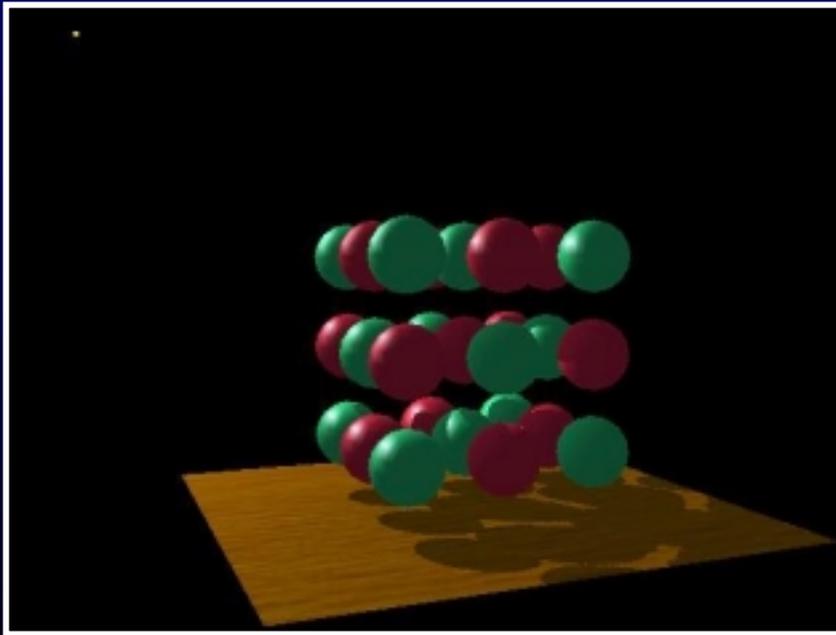
*A fairly complex scene with shadows*

*the point  
light source*

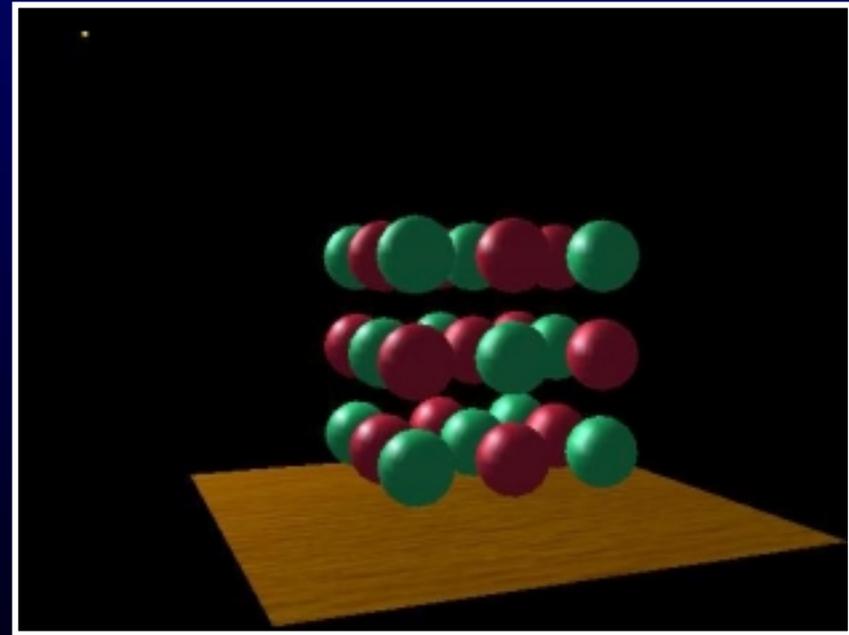


# Visualizing the Shadow Mapping Technique (2)

*Compare with and without shadows*



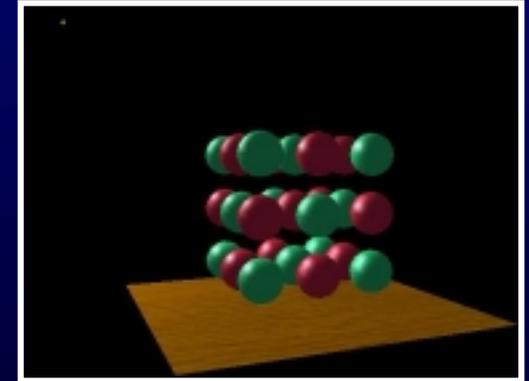
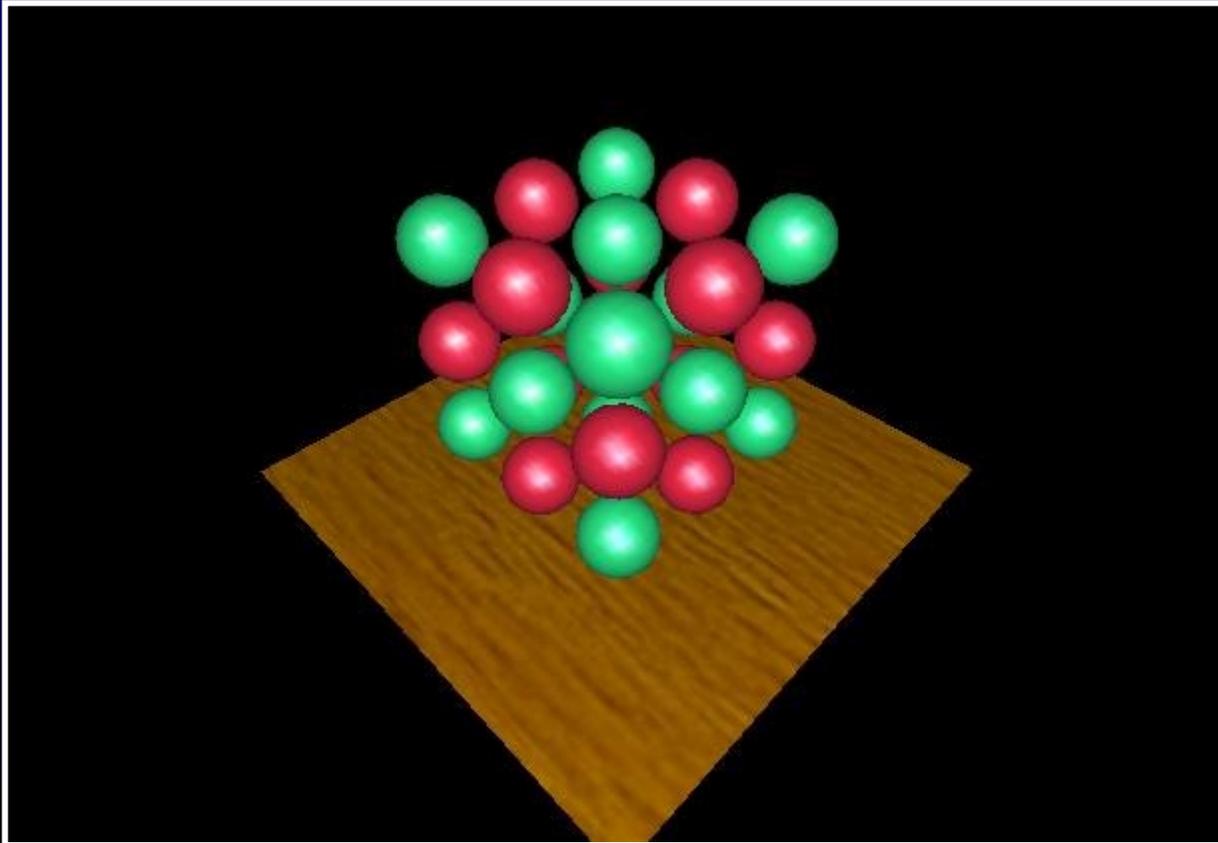
*with shadows*



*without shadows*

# Visualizing the Shadow Mapping Technique (3)

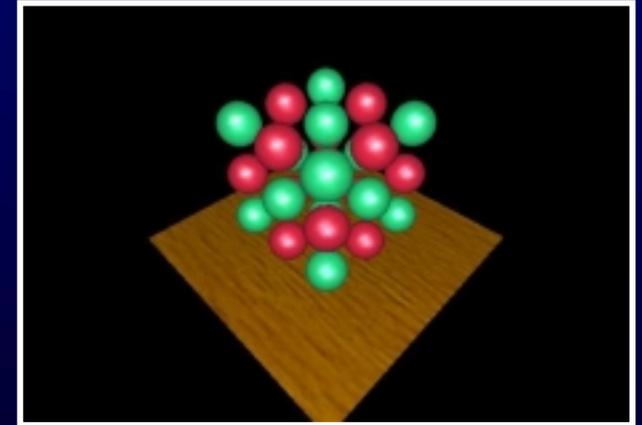
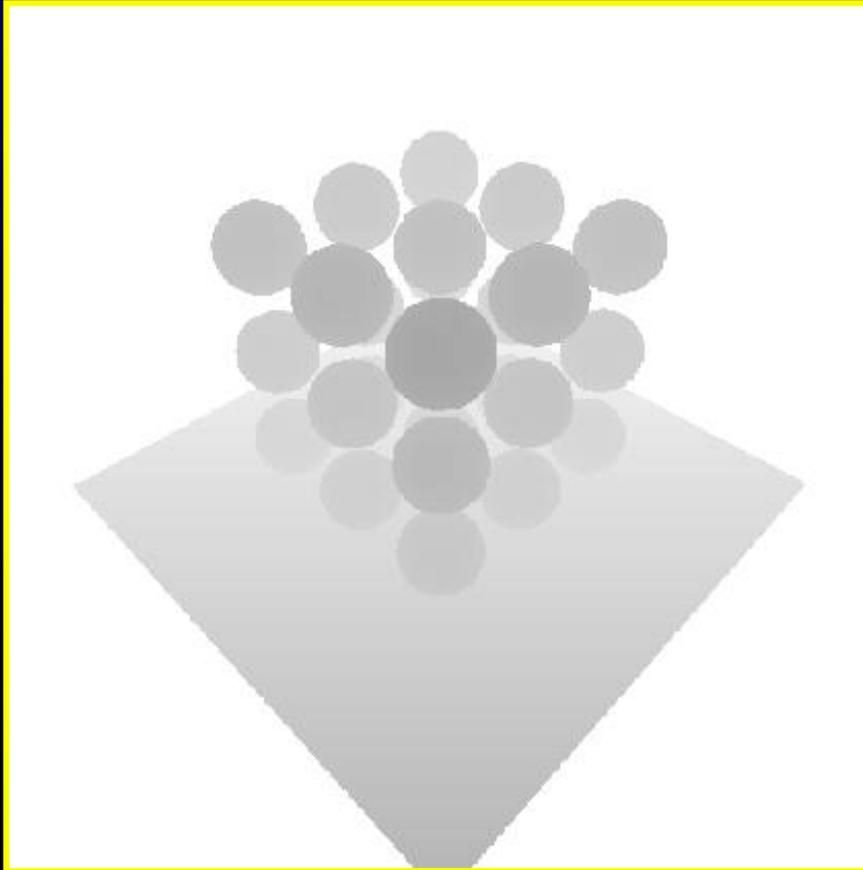
*The scene from the light's point-of-view*



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

# Visualizing the Shadow Mapping Technique (4)

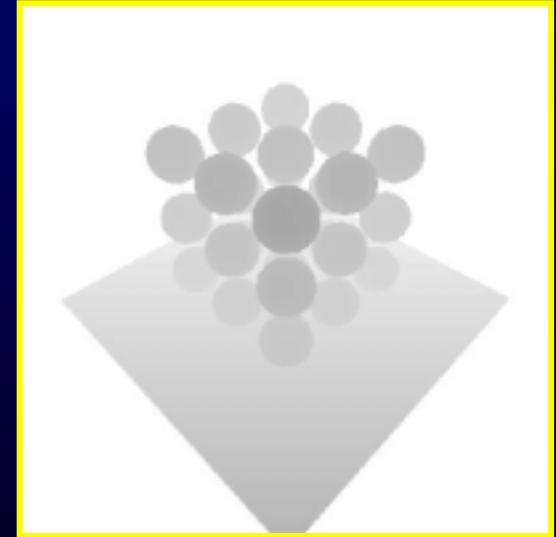
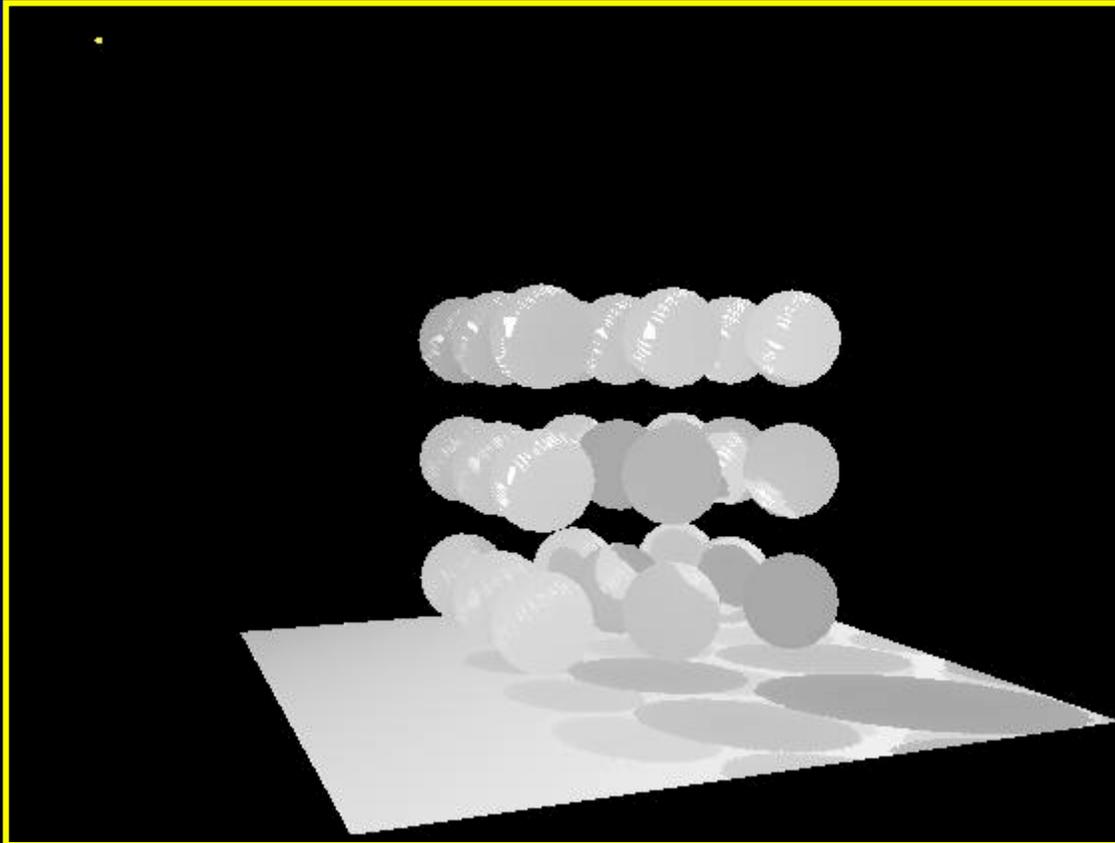
*The depth buffer from the light's point-of-view*



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

# Visualizing the Shadow Mapping Technique (5)

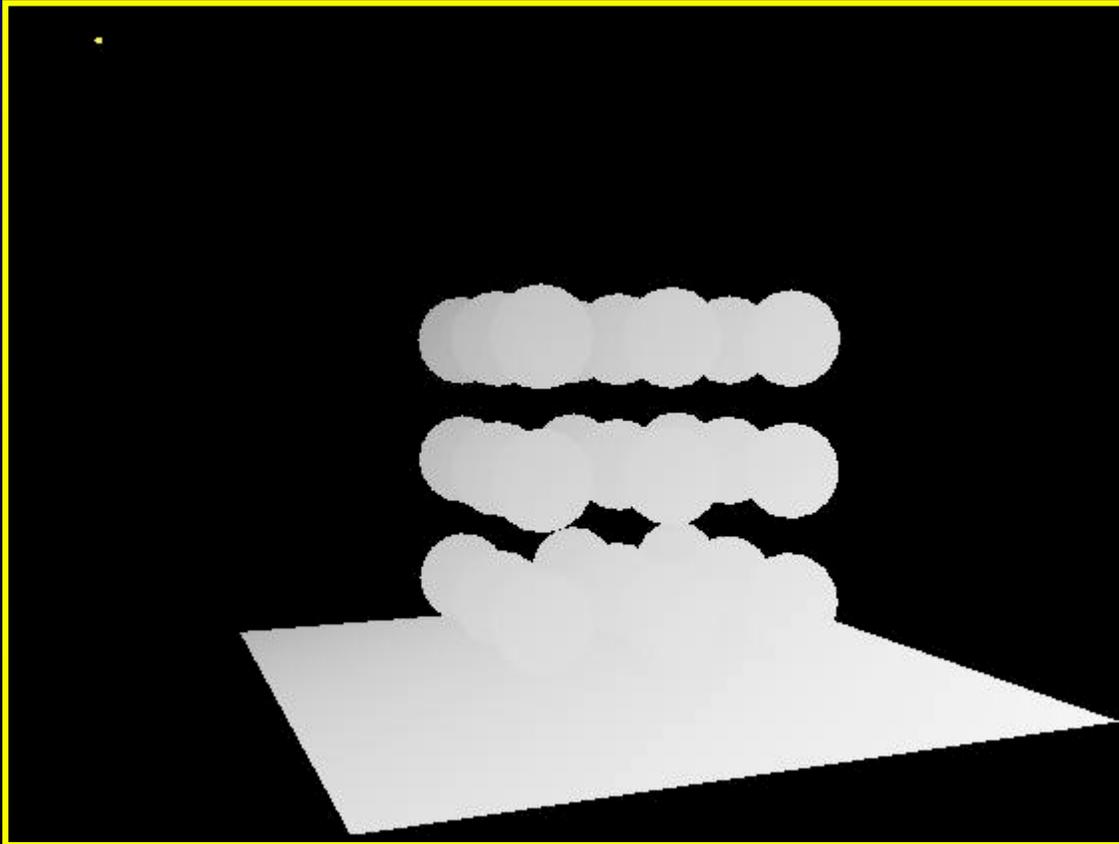
*Projecting the depth map onto the eye's view*



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

# Visualizing the Shadow Mapping Technique (6)

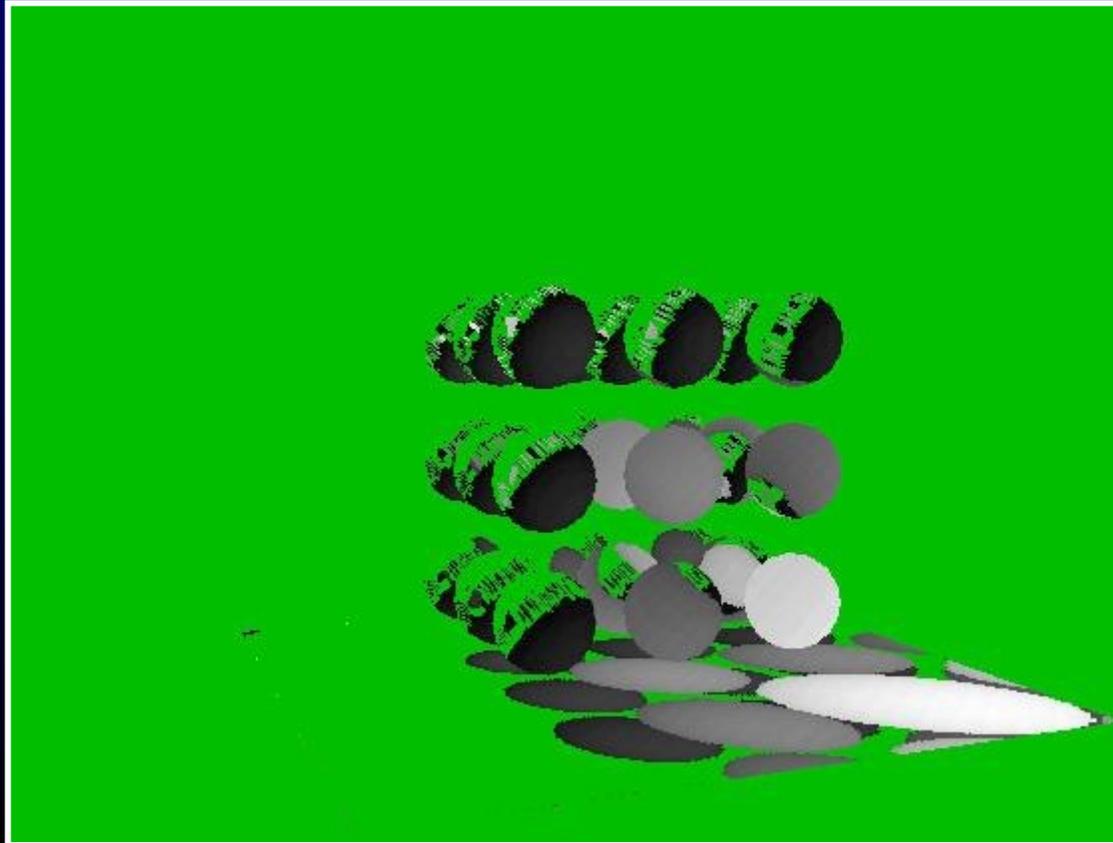
*Projecting light's planar distance onto eye's view*



# 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 approximately equal*

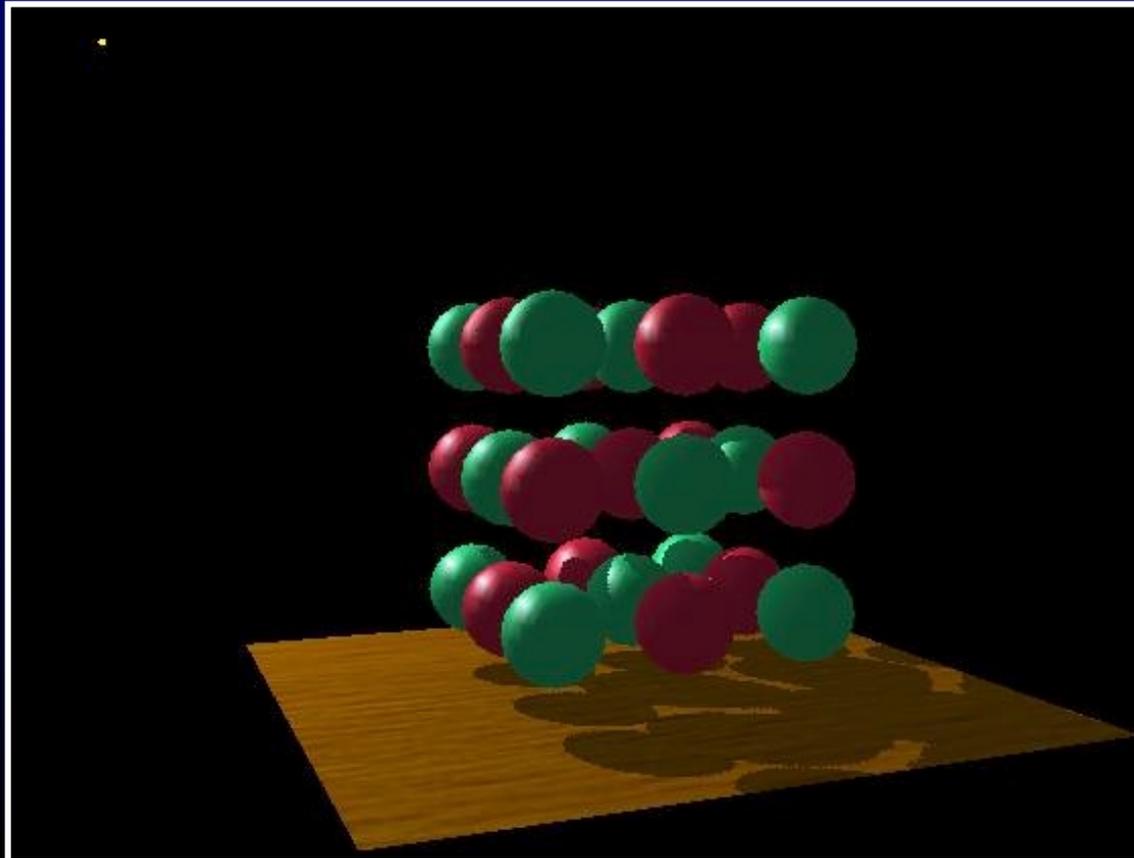


*Non-green is where shadows should be*

# 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*

# Construct Light View Depth Map

---



## *Realizing the theory in practice*

- Constructing the depth map
  - use existing hardware depth buffer
  - 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 (more on this later)

# Render Scene and Access the Depth Texture



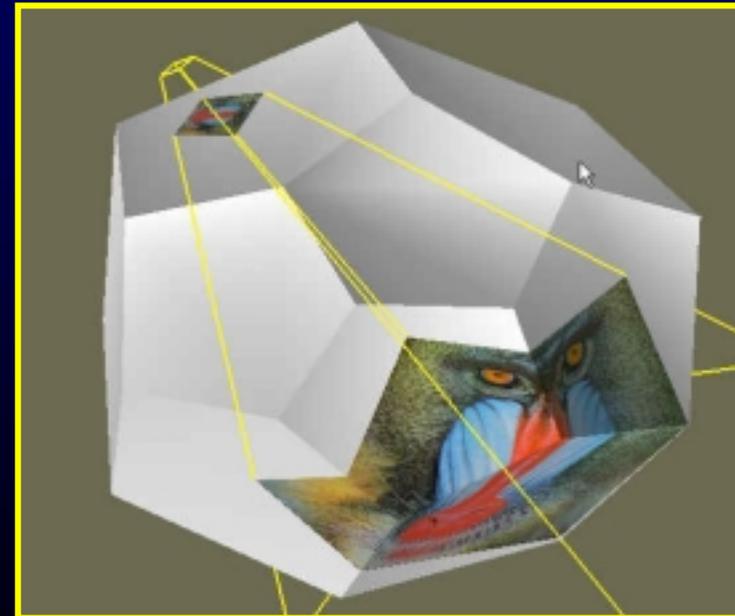
## *Realizing the theory in practice*

- 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 [99]*

# 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
  - $(s, t, r, q) \rightarrow (s/q, r/q, t/q)$
  - Similar to homogeneous clip coordinates where  $(x, y, z, w) = (x/w, y/w, z/w)$
- Idea is to have  $(s/q, r/q, t/q)$  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

# Projective Texturing

## Multitexturing

---



### *An aside about projective multi-texturing*

- Multi-texturing is easier if all texture units are required only to be perspective-correct
  - just requires a single hyperbolic interpolator (effectively shares a single divider among multiple texture units)
  - because  $1/w$  is the same for all texture units
- But multi-textured projective textures is harder
  - each texture unit could have a different  $q$
  - therefore a different  $q/w$  per texture unit

# NVIDIA's Projective Texturing Story

---



## *Different generations differ*

- TNT generation has a single shared hyperbolic interpolator
  - independently projected dual textures do not work
  - not enough gates for dual-projective in TNT timeframe
- GeForce generation has distinct q/w hyperbolic interpolators for both texture units (bigger gate budget buys correctness)
  - dual projective textures works
- Not sure what other vendors do

# Back to the Shadow Mapping Discussion . . .



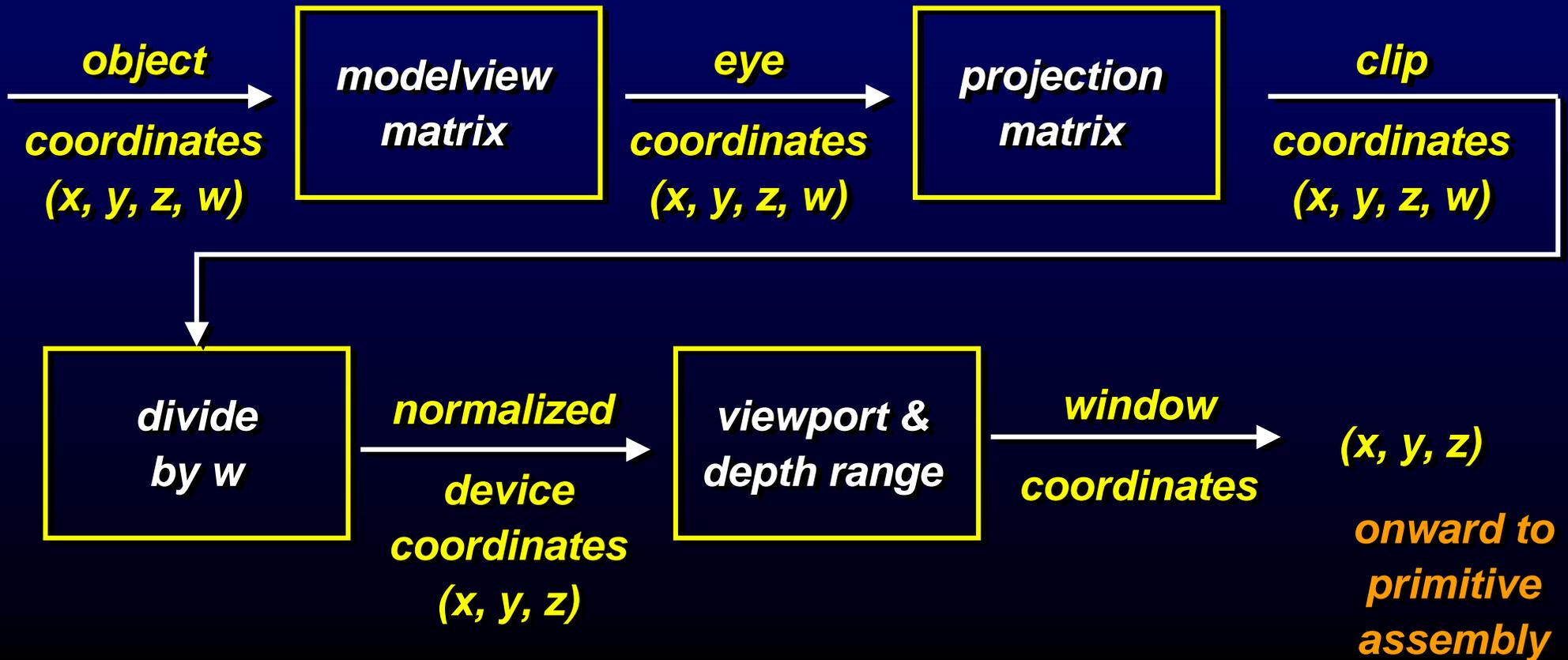
## *Assign light-space texture coordinates via texgen*

- 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



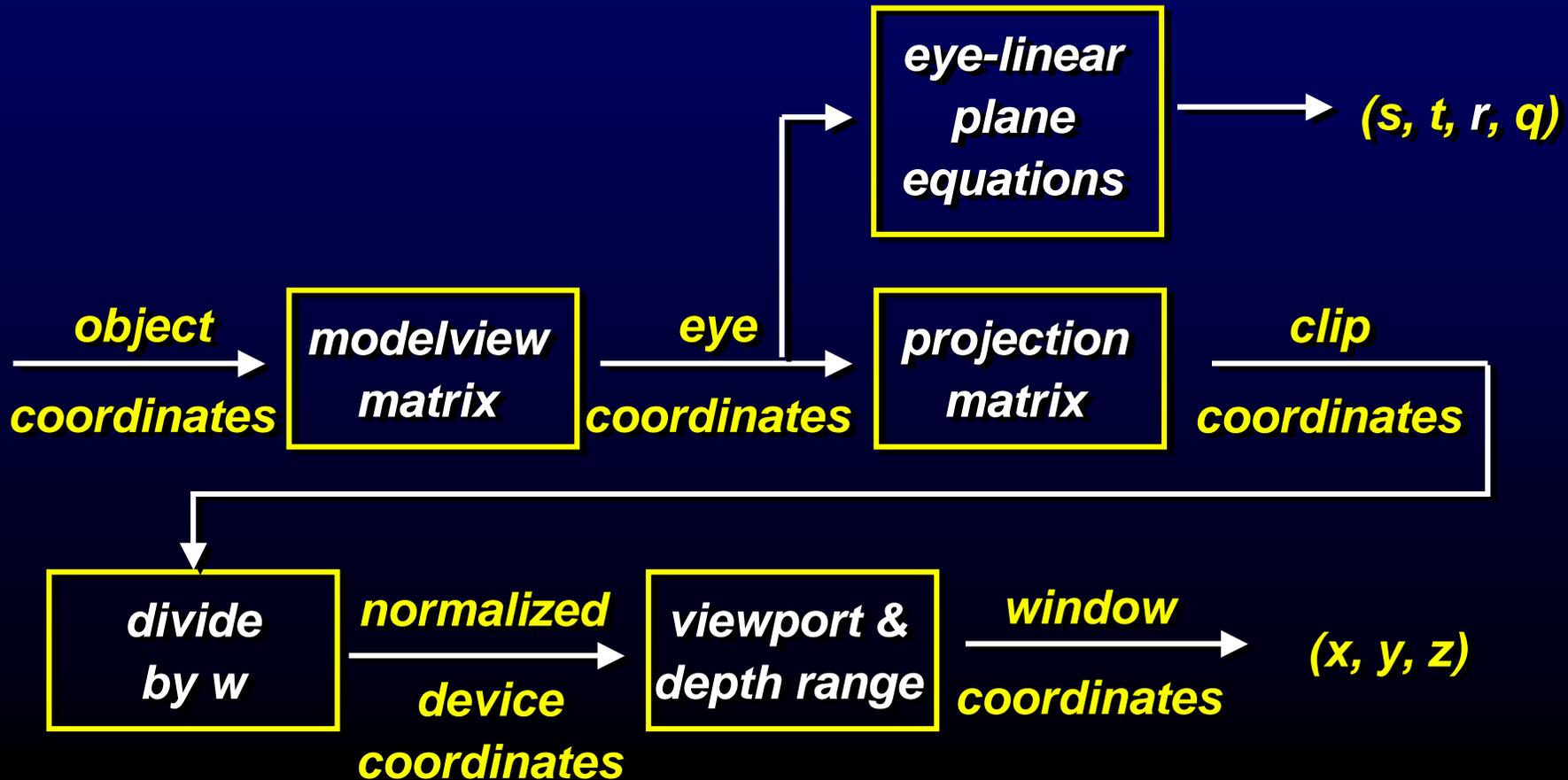
*From object coordinates to window coordinates*



# 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 plane equation is transformed by current inverse modelview matrix (a very handy thing for us)*

# 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 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



## *SGI RealityEngine and InfiniteReality 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  $(s/q, t/q)$

# 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

# Dual-texture Shadow Mapping Approach



## *Constructing the depth map texture*

- Render scene from the light view (can disable color writes)
  - use *glPolygonOffset* to bias depth values to avoid surfaces shadowing themselves in subsequent shadow test pass
  - perform bias during depth map construct instead of during shadow testing pass so bias will be in depth buffer space
- Read back depth buffer with *glReadPixels* as unsigned bytes
- Load same bytes into *GL\_INTENSITY8* texture via *glTexImage2D*

# Dual-texture Shadow Mapping Approach



## *Depth map texture issues*

- limited to 8-bit precision
  - not a lot of precision of depth
  - more about this issue later
- un-extended OpenGL provides no direct depth copy
  - cannot copy depth buffer to a texture directly
  - must *glReadPixels*, then *glTexImage2D*

# Dual-texture Shadow Mapping Approach



## *Two-pass shadow determination*

- 1st pass: draw everything shadowed
  - render scene with light disabled -or- dimmed substantially and specular light color of zero
  - with depth testing enabled
- 2nd pass: draw unshadowed, rejecting shadowed fragments
  - use `glDepthFunc(GL_EQUAL)` to match 1st pass pixels
  - enable the light source, un-rejected pixels = *unshadowed*
  - use dual-texture as described in subsequent slides

# Dual-texture Shadow Mapping Approach



## *Dual-texture configuration*

- 1st texture unit
  - bind to 2D texture containing light's depth map texture
  - intensity texture format (same value in RGB and alpha)
- 2nd texture unit
  - bind to 1D texture containing a linear ramp from 0 to 1
  - maps S texture coordinate in  $[0, 1]$  range to intensity value in  $[0, 1]$  range

# Dual-texture Shadow Mapping Approach

## *Texgen Configuration*

- 1st texture unit using 2D texture
  - generate (s/q, t/q) to access depth map texture, ignore R

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

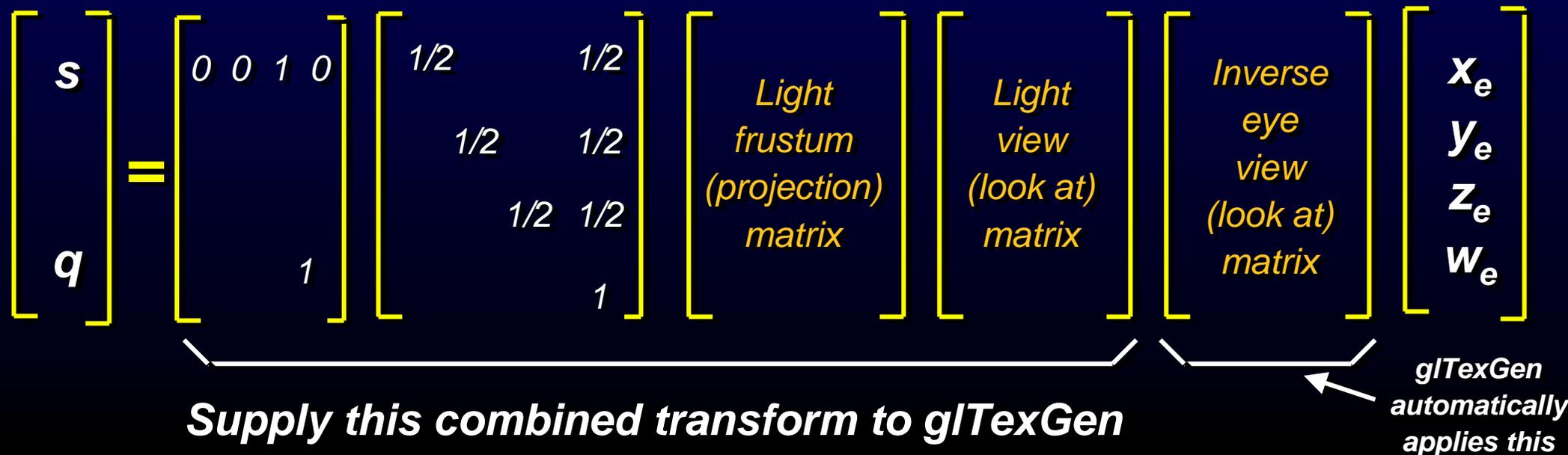
**Supply this combined transform to glTexGen**

glTexGen  
automatically  
applies this

# Dual-texture Shadow Mapping Approach

## *Texgen Configuration*

- 2nd texture unit using 1D texture
  - generate Z planar distance in S, flips what R is into S



# Dual-texture Shadow Mapping Approach



## *Texture environment (texenv) configuration*

- Compute the difference between **Tex0** from **Tex1**
  - un-extended OpenGL texenv cannot subtract
- But can use standard *EXT\_texture\_env\_combine* extension
  - add signed operation
  - compute fragment alpha as
$$\text{alpha}(\mathbf{Tex0}) + (1 - \text{alpha}(\mathbf{Tex1})) - 0.5$$
  - result is greater or equal to 0.5 when **Tex0**  $\geq$  **Tex1**  
result is less than 0.5 when **Tex0**  $<$  **Tex1**

# Dual-texture Shadow Mapping Approach

## *Texture environment (texenv) specifics*

```
glActiveTextureARB(GL_TEXTURE0_ARB);
```

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
```

```
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_REPLACE);
```

```
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE0_RGB_EXT, GL_PRIMARY_COLOR_EXT);
```

```
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND0_RGB_EXT, GL_SRC_COLOR);
```

```
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_ALPHA_EXT, GL_REPLACE);
```

```
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE0_ALPHA_EXT, GL_TEXTURE);
```

```
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND0_ALPHA_EXT, GL_SRC_ALPHA);
```

```
glActiveTextureARB(GL_TEXTURE1_ARB);
```

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
```

```
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_REPLACE);
```

```
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE0_RGB_EXT, GL_PREVIOUS_EXT);
```

```
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND0_RGB_EXT, GL_SRC_COLOR);
```

```
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_ALPHA_EXT, GL_ADD_SIGNED_EXT);
```

```
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE0_ALPHA_EXT, GL_PREVIOUS_EXT);
```

```
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND0_ALPHA_EXT, GL_SRC_ALPHA);
```

```
glTexEnvf(GL_TEXTURE_ENV, GL_SOURCE1_ALPHA_EXT, GL_TEXTURE);
```

```
glTexEnvf(GL_TEXTURE_ENV, GL_OPERAND1_ALPHA_EXT, GL_ONE_MINUS_SRC_ALPHA);
```

# Dual-texture Shadow Mapping Approach



## *Post-texture environment result*

- RGB is lit color (lighting is enabled during second pass)
- Alpha is the biased difference of T0 and T1
  - unshadowed fragments have  $\alpha \geq 0.5$
  - shadowed fragments have an alpha of  $< 0.5$

# Dual-texture Shadow Mapping Approach



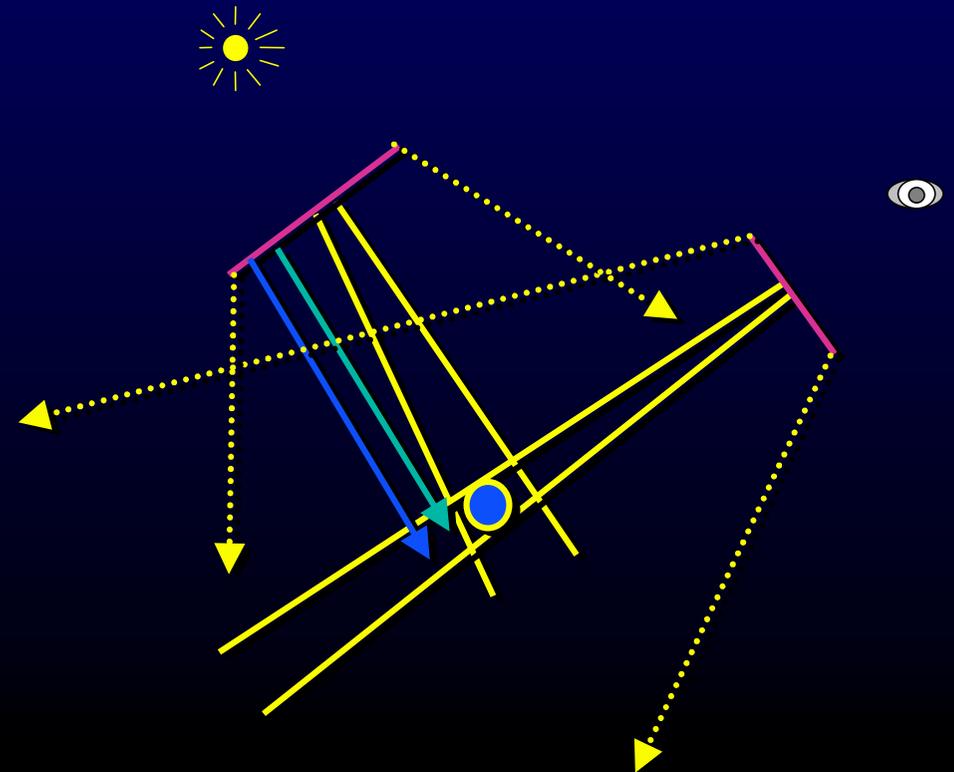
## *Next, reject shadowed fragments*

- shadowed or unshadowed depends on alpha value
  - less than 0.5 means shadowed
- use the alpha test to reject shadowed fragments
  - `glEnable(GL_ALPHA_TEST)`
  - `glAlphaFunc(GL_GREATER, 0.5)`

# Dual-texture Shadow Mapping Approach

## *Careful about self-shadowing*

- fragments are likely to shadow themselves
  - surface casting shadow must not shadow itself
  - “near equality” common when comparing Tex0 and Tex1



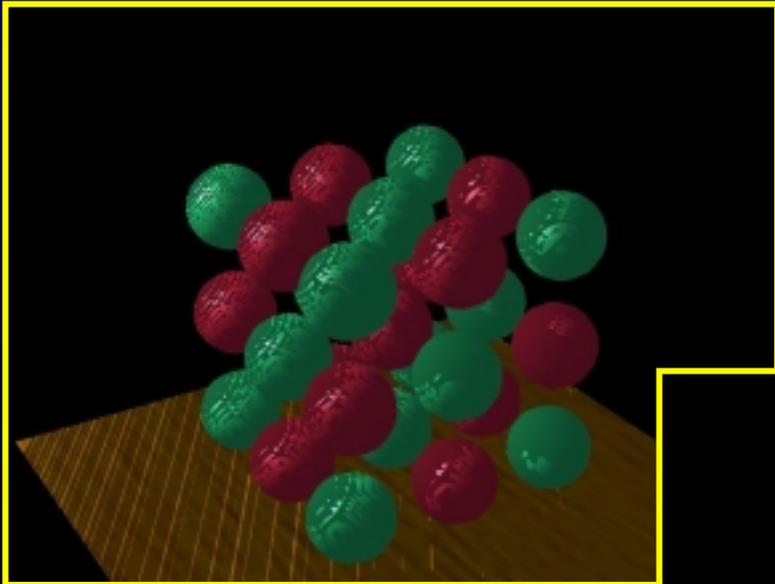
# Dual-texture Shadow Mapping Approach

## *Biasing values in depth map helps*

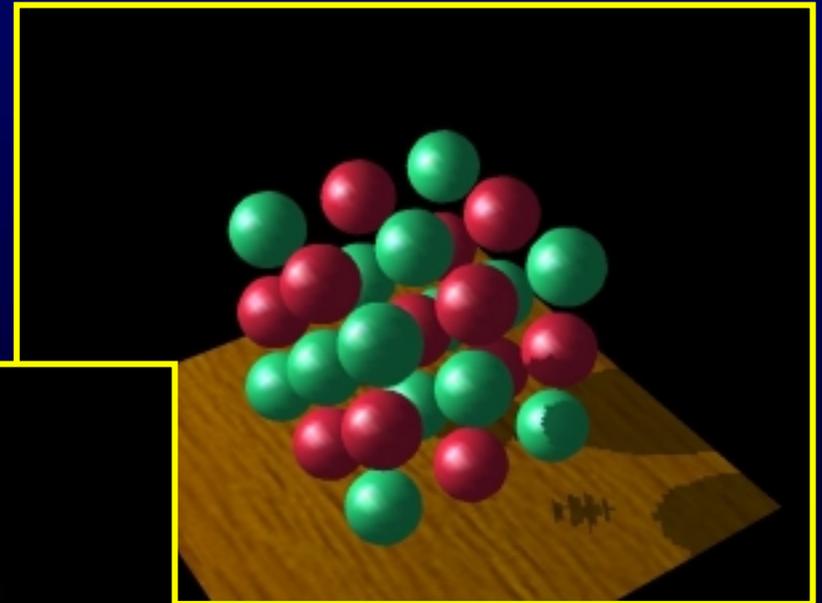
- recall *glPolygonOffset* suggestion during the depth map construction pass
- this bias should be done during depth map construction
  - biases in the texgen transform do not work
  - problem is depth map has non-linear distribution due to projective frustum
- polygon offset scale keeps edge-on polygons from self-shadowing

# Depth Map Bias Issues

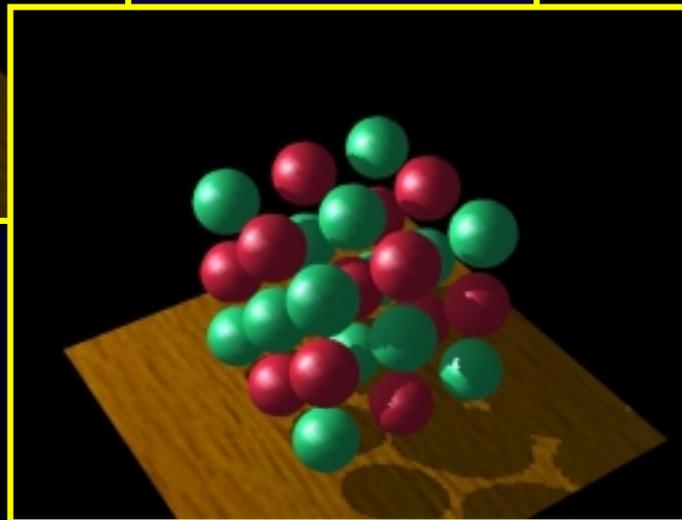
## *How much polygon offset bias depends*



*Too little bias,  
everything begins to  
shadow*



*Too little bias, shadow  
starts too far back*



*Just right*

# Selecting the Depth Map Bias



## *Not that hard*

- Usually the following works well
  - `glPolygonOffset(scale = 1.0, 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

# Dual-texture Shadow Mapping Precision

---

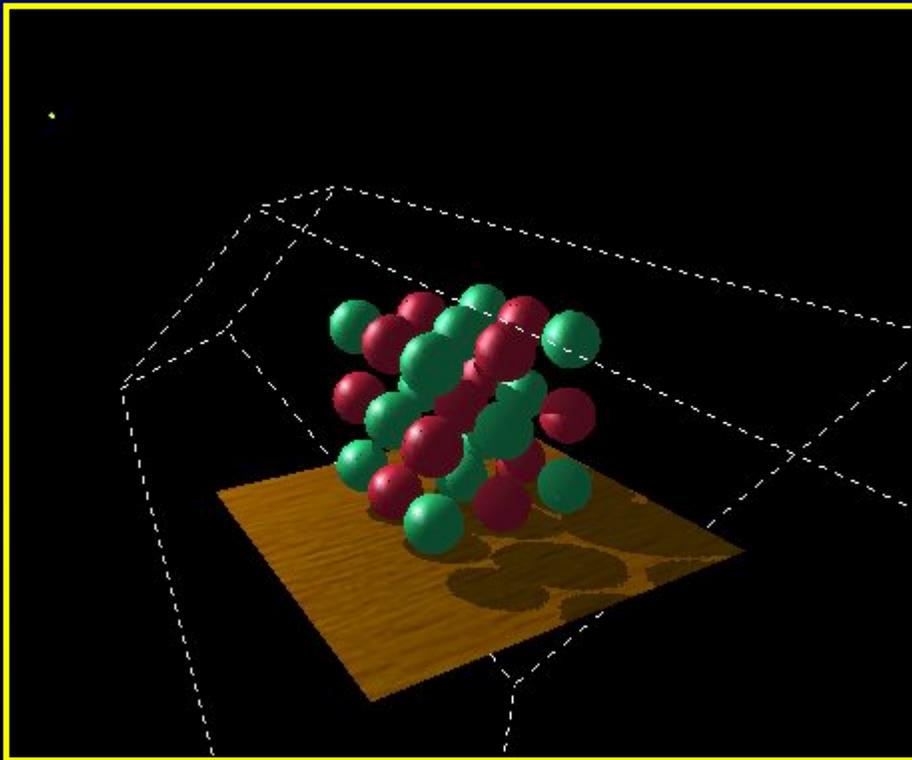


## *Is 8-bit precision enough?*

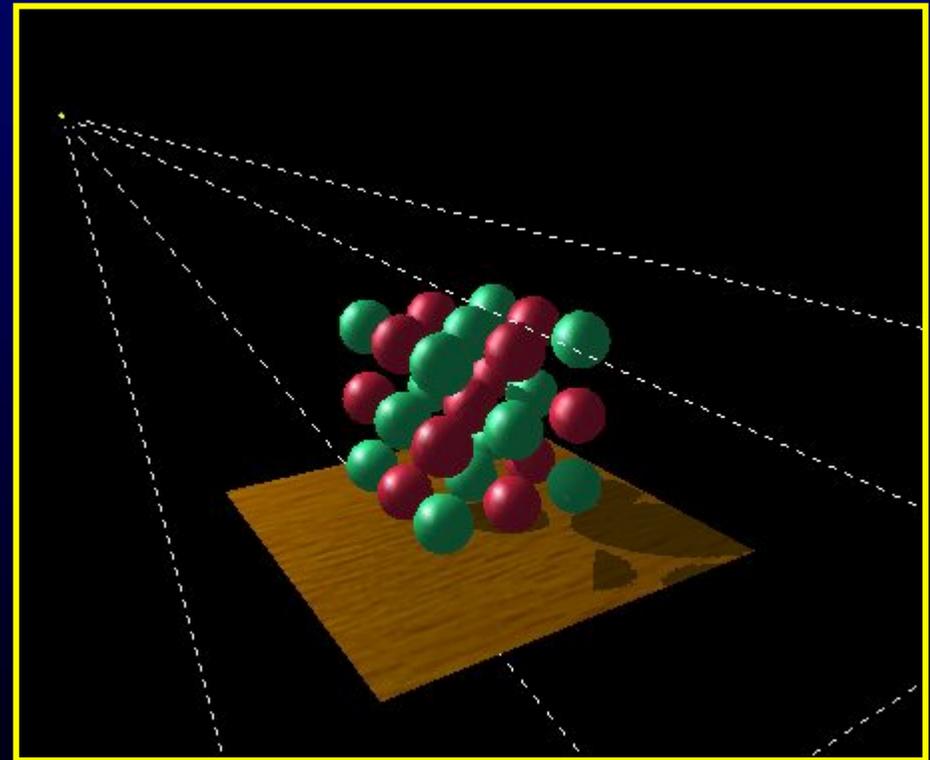
- yes, for some simple scenes
  - when the objects are relatively distant from the light, but still relatively close together
- no, in general
  - an 8-bit depth buffer is not enough depth discrimination
  - and the precision is badly distributed because of perspective

# Dual-texture Shadow Mapping Precision

## *Conserving your 8-bit depth map precision*



*Frustum confined to objects of interest*



*Frustum expanded out considerably  
breaks down the shadows*

# Improving Depth Map Precision



## *Use linear depth precision [Wolfgang 99]*

- During depth map construction
  - generate S texture coordinate as eye planar Z distance scaled to [0, 1] range
  - lookup S in identity 1D intensity texture
  - write texture result into color frame buffer
  - still using standard depth testing
  - read alpha (instead of depth) and load it in depth map texture
  - alpha will have linear depth distribution (better!)

# Improving Depth Map Precision

## ***More hardware color component precision***

- high-end workstations support more color precision
  - SGI's InfiniteReality, RealityEngine, and Octane workstations support 12-bit color component precision
- but no high precision color buffers in consumer 3D space
  - consumer 3D designs too tied to 32-bit memory word size of commodity RAM
  - and overkill for most consumer applications anyway

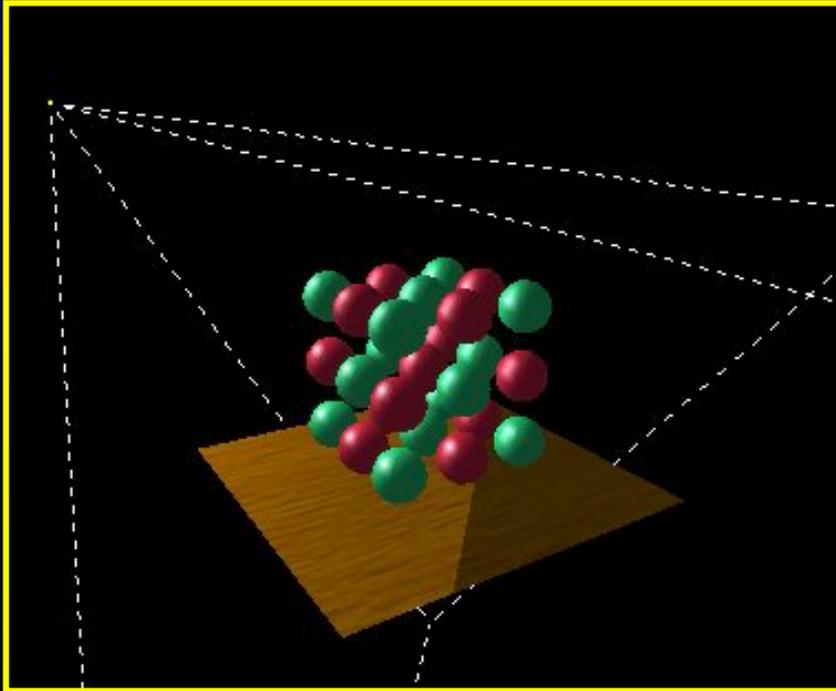
# Improving Depth Map Precision

## *Use multi-digit comparison*

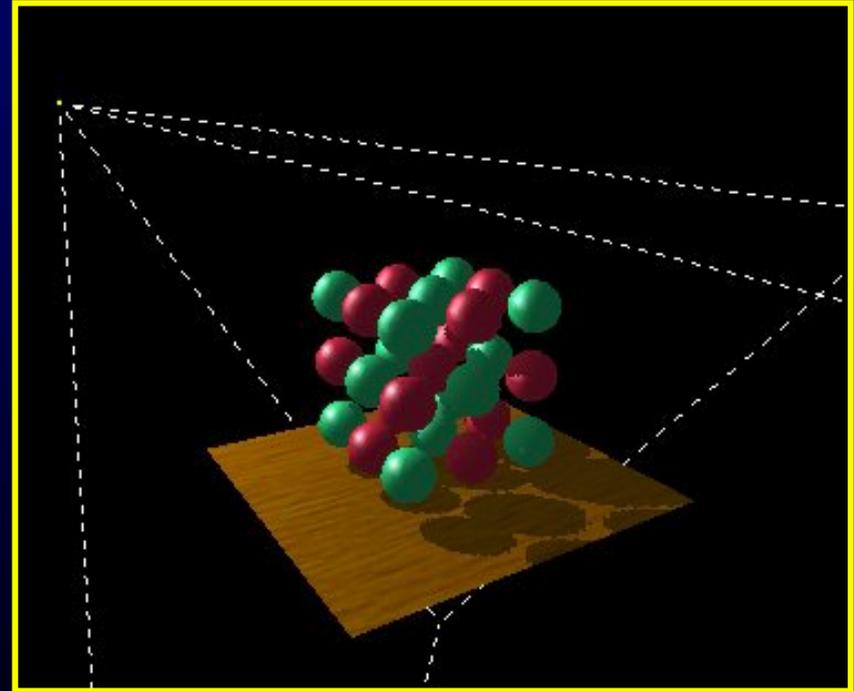
- fundamental shadow determination operation is a comparison
  - comparisons (unlike multiplies or other operations) are easy to extend to higher precision
- think about comparing two 2-digit numbers: 54 and 82
  - 54 is less than 82 simply based on the first digit ( $5 < 8$ )
  - only when most-significant digits are equal do you need to look at subsequent digits

# More Precision Allows Larger Lights Frustums

*Compare 8-bit to 16-bit precision for large frustum*



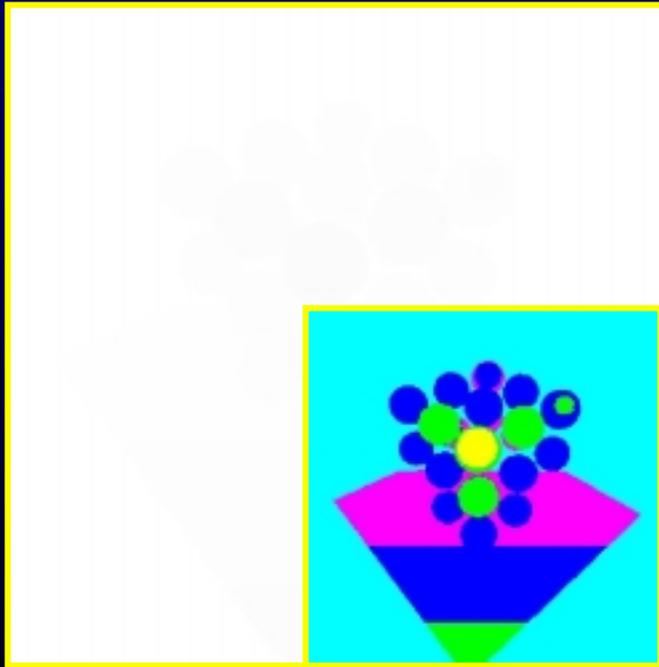
*8-bit: Large frustum breaks down the shadows, not enough precision*



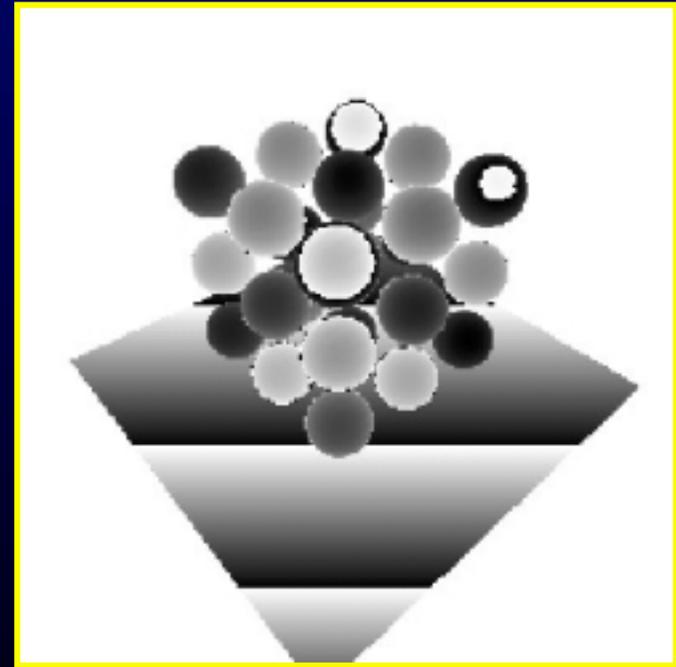
*16-bit: Shadow looks just fine*

# Why Extra Precision Helps

*Where the precision is for previous images*



*Most significant 8 bits of the depth map,  
pseudo-color inset magnifies variations*



*Least significant 8 bits of the depth map,  
here is where the information is!*

# GeForce/Quadro Precision Extension



## *Application of multi-digit comparison idea*

- Read back depth buffer as 16-bit unsigned short values
- Load these values into GL\_LUMINANCE8\_ALPHA8 texture
  - think of depth map as two 8-bit digits
- Two comparison passes
- Uses NV\_register\_combiners extension
  - signed math and mux'ing helps
  - enough operations to test equality and greater/less than

# GeForce/Quadro Precision Extension



## *Multi-digit comparison passes*

- during clear, clear stencil buffer to 0
- 1st pass draws unshadowed scene as before
- 2nd pass draws unshadowed
  - $\alpha = \text{bigDigit}(\text{Tex0}) < \text{bigDigit}(\text{Tex1})$
  - alpha test with `glAlphaFunc(GL_GREATER, 0.0)`
  - and write 1 into stencil buffer when alpha test passes
- needs 3rd pass for when  $\text{bigDigit}(\text{Tex0}) = \text{bigDigit}(\text{Tex1})$

# GeForce/Quadro Precision Extension



## *Third pass picks up the extra 8 bits of precision*

- Use NV\_register\_combiners to assign alpha as follows
  - if  $\text{bigDigit}(\text{Tex1}) > \text{bigDigit}(\text{Tex0})$  then  
alpha = 0
  - else  
alpha =  $\text{littleDigit}(\text{Tex0}) - \text{littleDigit}(\text{Tex1})$
- Use alpha test with `glAlphaFunc(GL_GREATER, 0.0)`
- Also reject fragment if the stencil value is 1
  - meaning the 2nd pass already updated the pixel

# 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

# Issues with Shadow Mapping (1)



## *Not without its problems*

- Prone to aliasing artifacts
  - “percentage closest” 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 loading 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

# Hybrid of Shadow Volumes and Shadow Mapping

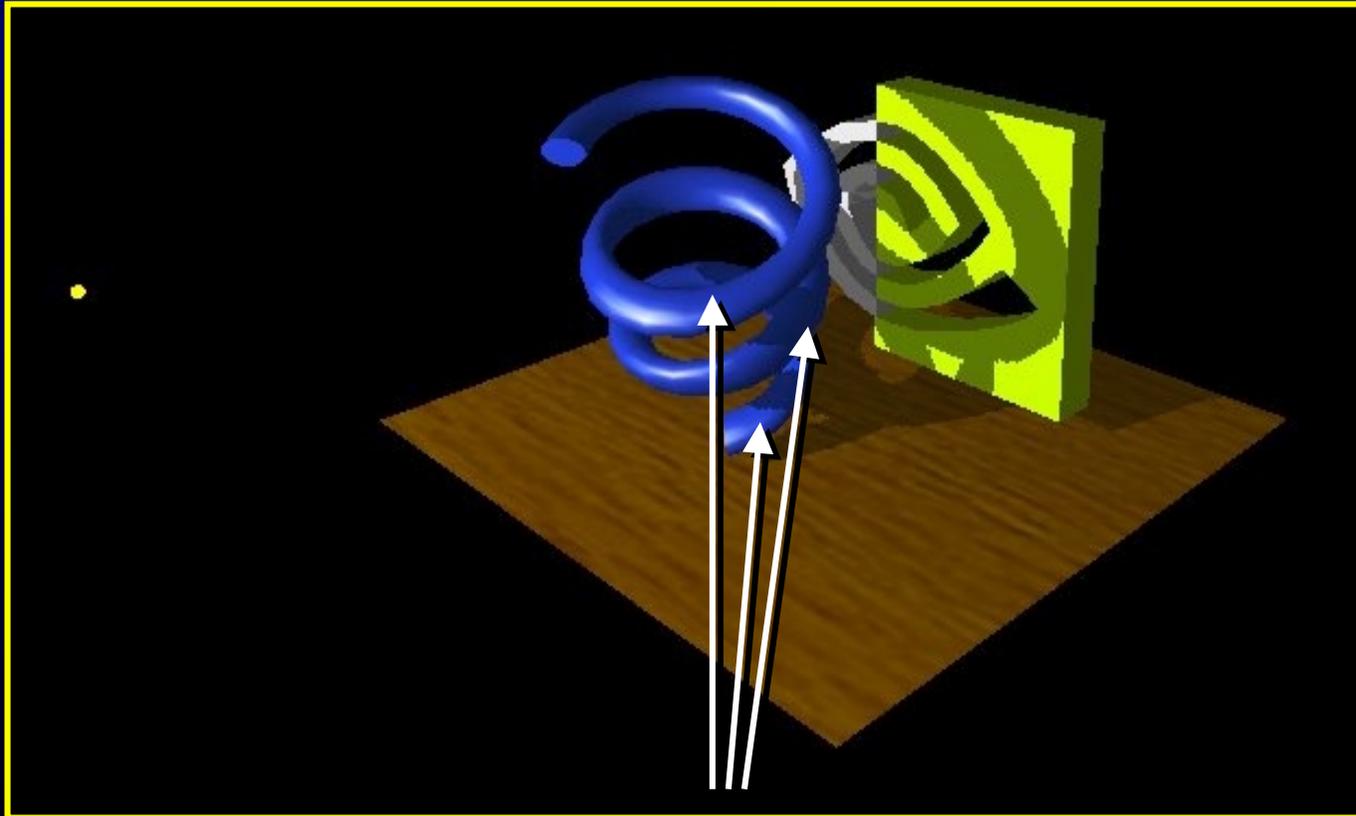


## *Very clever idea [McCool 98]*

- Render scene from light source with depth testing
- Read back the depth buffer
- Use computer vision techniques to reconstruct the shadow volume *geometry* from the depth buffer *image*
- Very reasonable results for complex scenes
- Only requires stencil
  - no multitexture and texture environment differencing required

# More Examples

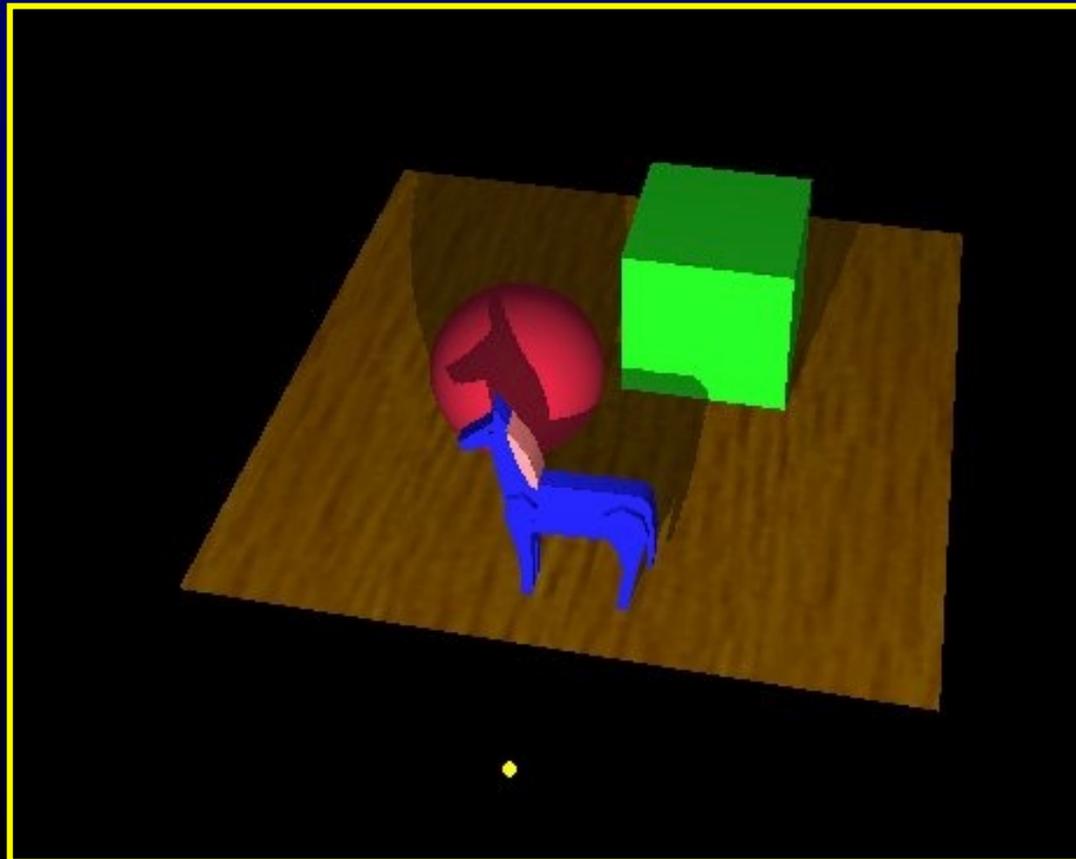
## *Smooth surfaces with object self-shadowing*



*Note object self-shadowing*

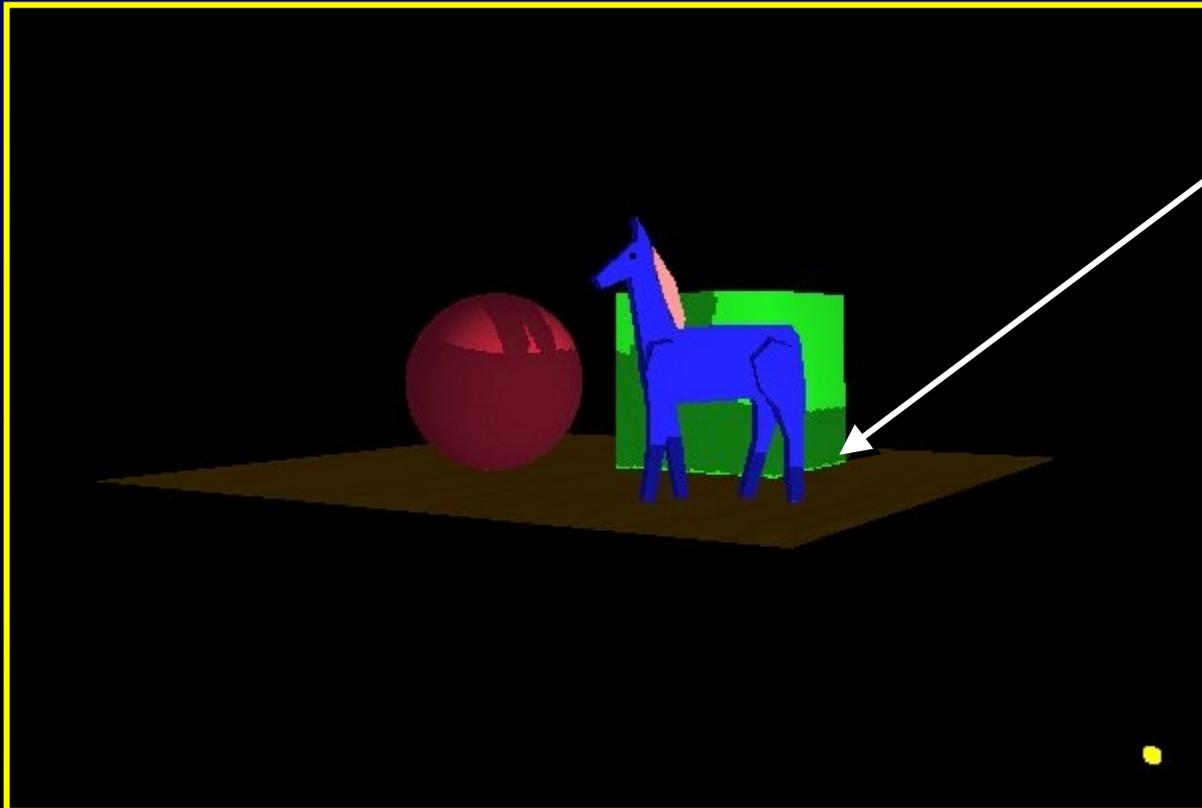
# More Examples

*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*

# Shadow Mapping Source Code



## *Find it on the NVIDIA web site*

- The source code
  - <http://www.nvidia.com/opengl/ShadowMap>
  - Works on TNT, GeForce, & Quadro
  - And vendors that support EXT\_texture\_env\_combine
- *NVIDIA OpenGL Extension Specifications*
  - documents EXT\_texture\_env\_combine and NV\_register\_combiners
  - <http://www.nvidia.com/opengl/openglspecs>

## *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