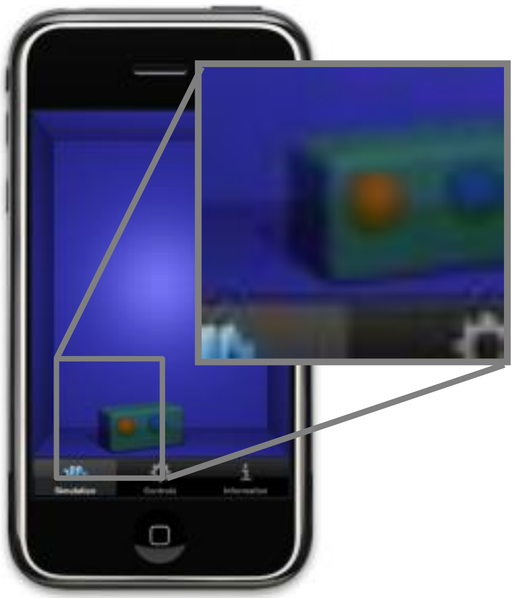
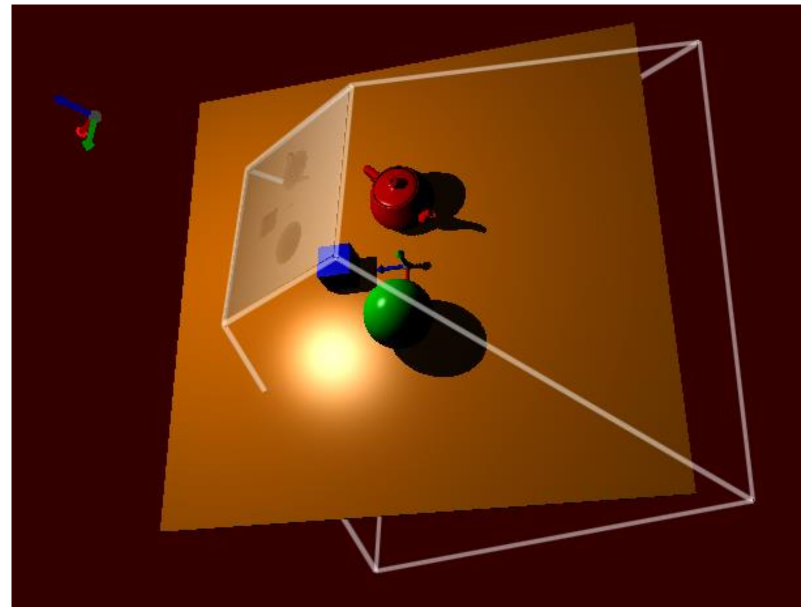
Shadows and Object Order Rendering

Projective Textures, Shadow Maps, and Cheap Shadows





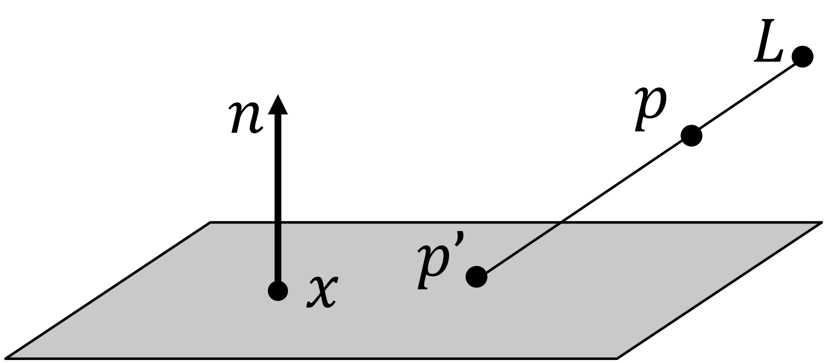
- Can do fake shadows by projecting geometry onto planar surfaces
- Texture coordinate generation to project textures onto surfaces (old style, before vertex and fragment programs)
  - Compare z values of visible surfaces to a z value as seen from light
- Can also use shadow volume technique to get pixel accurate sharp shadows.

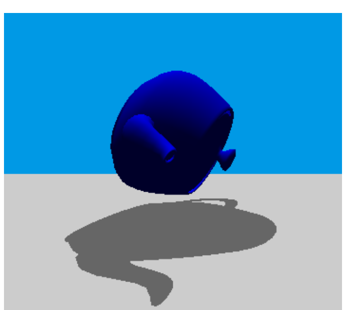
Cheap Shadow Projection

- Redraw geometry projected onto floor plane using a dark solid colour (i.e., *draw the shadows*)
  - Only good for planar shadows in simple environments.
  - No self shadowing, but often very compelling.

Cheap Shadow Projection

- Derive a homogeneous transformation matrix that projects point P onto ground plane point P’
  - Plane defined with normal, n, and point on plane, x





From the plane equation, let

$D = -n^T x$ i.e., $n^T p' + D = 0.$

Write the parameterized ray from the light through p as

$$p' = L + t(p - L)$$

and solve for t such that p' is on the plane,

$$n^T(L + t(p - L)) + D = 0.$$

Solving for t gives $t = \frac{-(D+n^T L)}{n^T(p-L)}$.

Thus,

$$p' = L + \frac{-(D + n^T L)(p - L)}{n^T(p - L)}$$

We can tease this apart into a linear part, translation, and perspective divide…

$$p' = L + \frac{-(D + nTL)(p - L)}{n^T(p - L)}$$

$$p' = \frac{Ln^T(p - L)}{n^T(p - L)} + \frac{-(D + nTL)(p - L)}{n^T(p - L)}$$

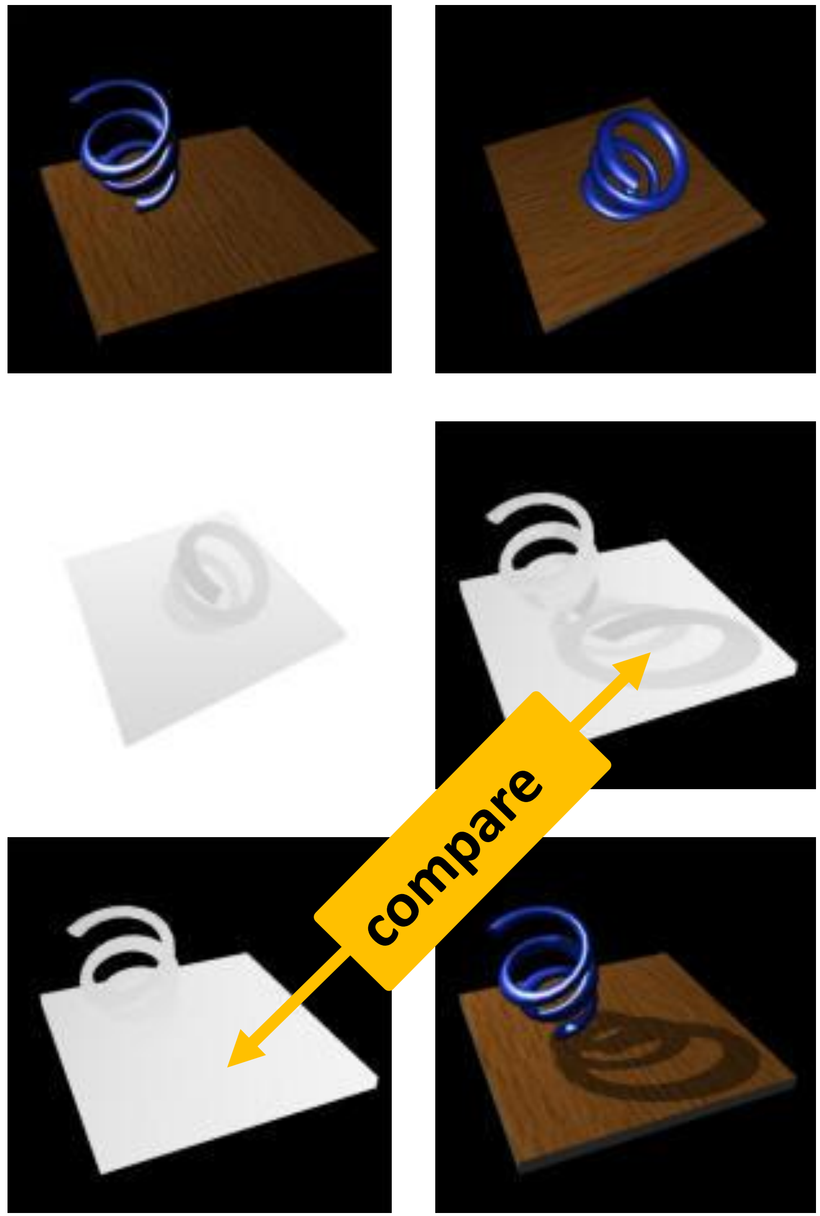
$$p' = \frac{Ln^T(-L) + (D + nTL)L}{n^T(p - L)} + \frac{Ln^Tp + (-D - n^TL)p}{n^T(p - L)}$$

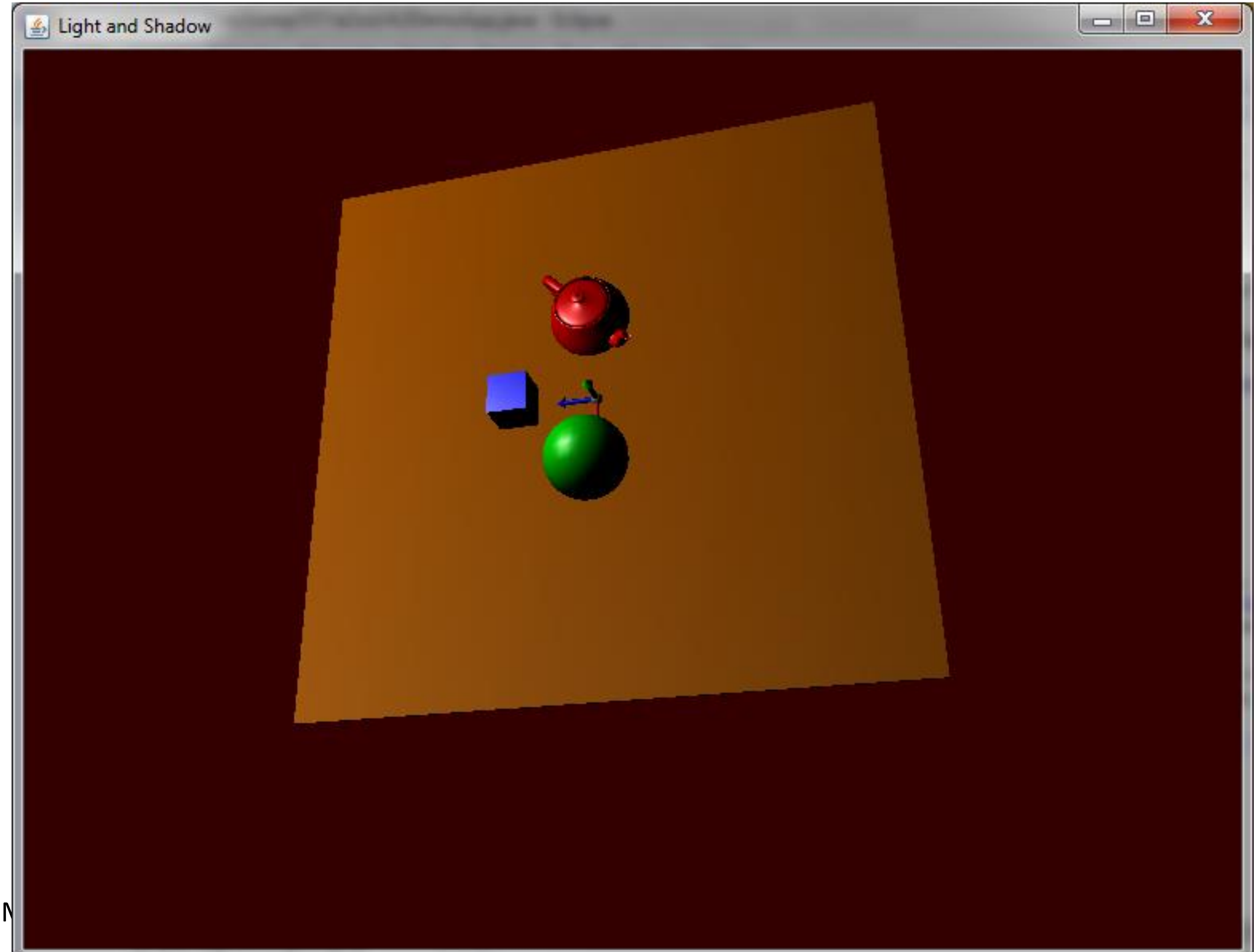
$$p' = \begin{pmatrix} Ln^T - (D + n^TL)I & DL \\ n^T & -n^TL \end{pmatrix} \begin{pmatrix} p \\ 1 \end{pmatrix}$$

Shadows

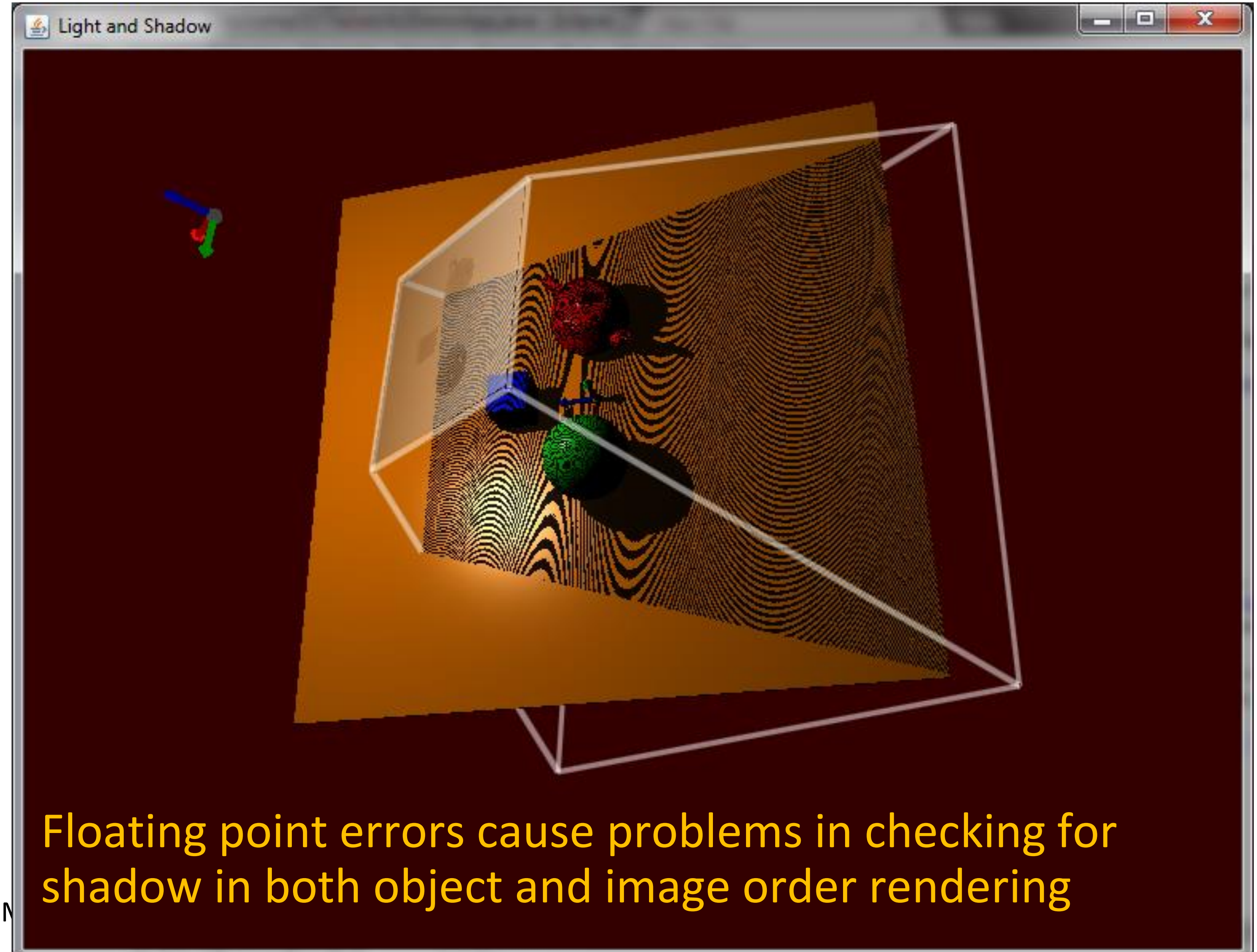
- Surface is only illuminated if nothing blocks its view of the light.
- Need to check if anything is occluding
  - The option commonly used in object order rendering is to draw the view from the light, and store information about the depth of the closest surfaces.
  - ***Shadow mapping***

Shadow Maps


eye view
light view
light depth
light depth in
eye view
compare
eye view depth
eye view
with shadows

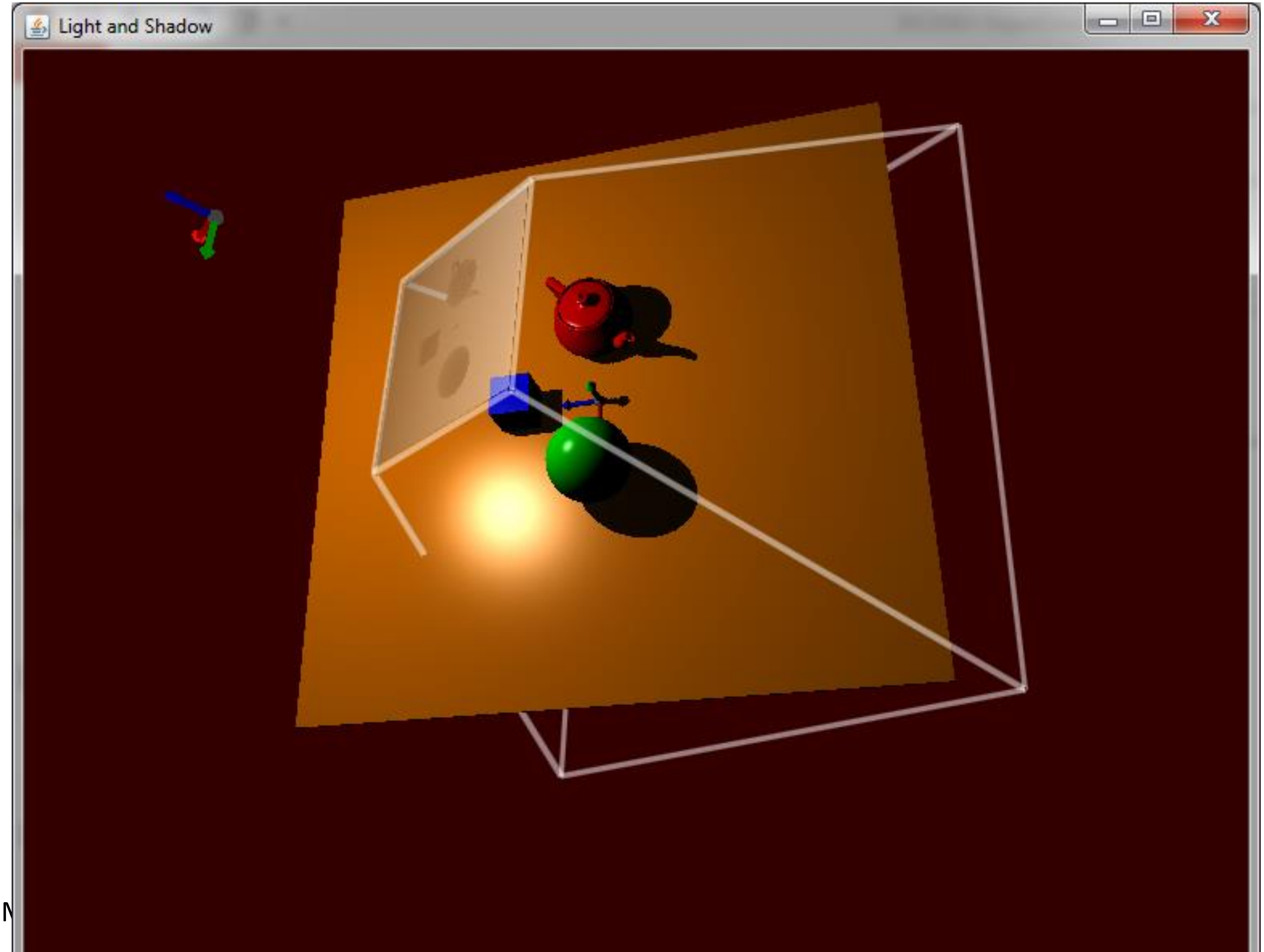

Light and Shadow

What's going on? → Self shadowing



Floating point errors cause problems in checking for shadow in both object and image order rendering

Add small offset to comparison…



GLSL Lighting Computations

chapter 9 of orange book (Emulating OpenGL Fixed Functionality)

- Orange book shows how the vertex program can compute the eye coordinate position of the vertex.
- In per fragment lighting, the ***vertex program*** computes the surface fragment position in camera coordinates and stores it in a ***varying*** vec3, to be used in the ***fragment program***.

```
varying vec3 v; // surface fragment location in camera
…
v = vec3( gl_ModelViewMatrix * gl_Vertex );
```

- Varying quantities are set at the vertex stage, and interpolated quantities are available at the fragment stage
  - Recall slides on interpolation and rasterization.
  - Can also interpolate normals for Phong shading.

GLSL Lighting Computations

chapter 9 of orange book (Emulating OpenGL Fixed Functionality)

- The components of vectors are more commonly accessed with a field structure rather than array syntax, e.g., v.xyzw v.stqr v.rgba are valid for getting all components.
- You can ***swizzle*** entries and take portions as you see fit
  - v.yx is a vec2 with x and y reversed,
  - v.xxx is a vec3 formed by taking the x value of v repeated 3 times.
- Chapter 3 of the orange book talks more about data types and conversions.
- Typically, casts and conversions do the right thing, but you'll perhaps want to be sure!

GLSL Lighting Computations

chapter 9 of orange book (Emulating OpenGL Fixed Functionality)

```
varying vec3 v; // surface fragment location in camera coordinates
varying vec3 n; // normal of fragment in camera coordinates
```

Given the vertex location in the camera, a diffuse lighting component can be computed by computing a difference, a normalization, a dot product, and then clamp the result (in the ***fragment program***).

```
vec3 L = normalize( gl_LightSource[0].position.xyz - v );
vec4 Ld = gl_FrontLightProduct[0].diffuse * max(dot(n,L), 0.0);
Ld = clamp(Ld, 0.0, 1.0);
```

The front light product for light zero, is the component-wise product of kd and Id. Alternatively you could compute the product as

```
gl_MaterialParameters.diffuse * gl_LightSource[0].diffuse
```

See more with respect to the "built-in" *uniforms* that are available to implement functionality that uses fixed function pipeline settings.

Shadow Map lookup

```
uniform sampler2D shadowMap;
```

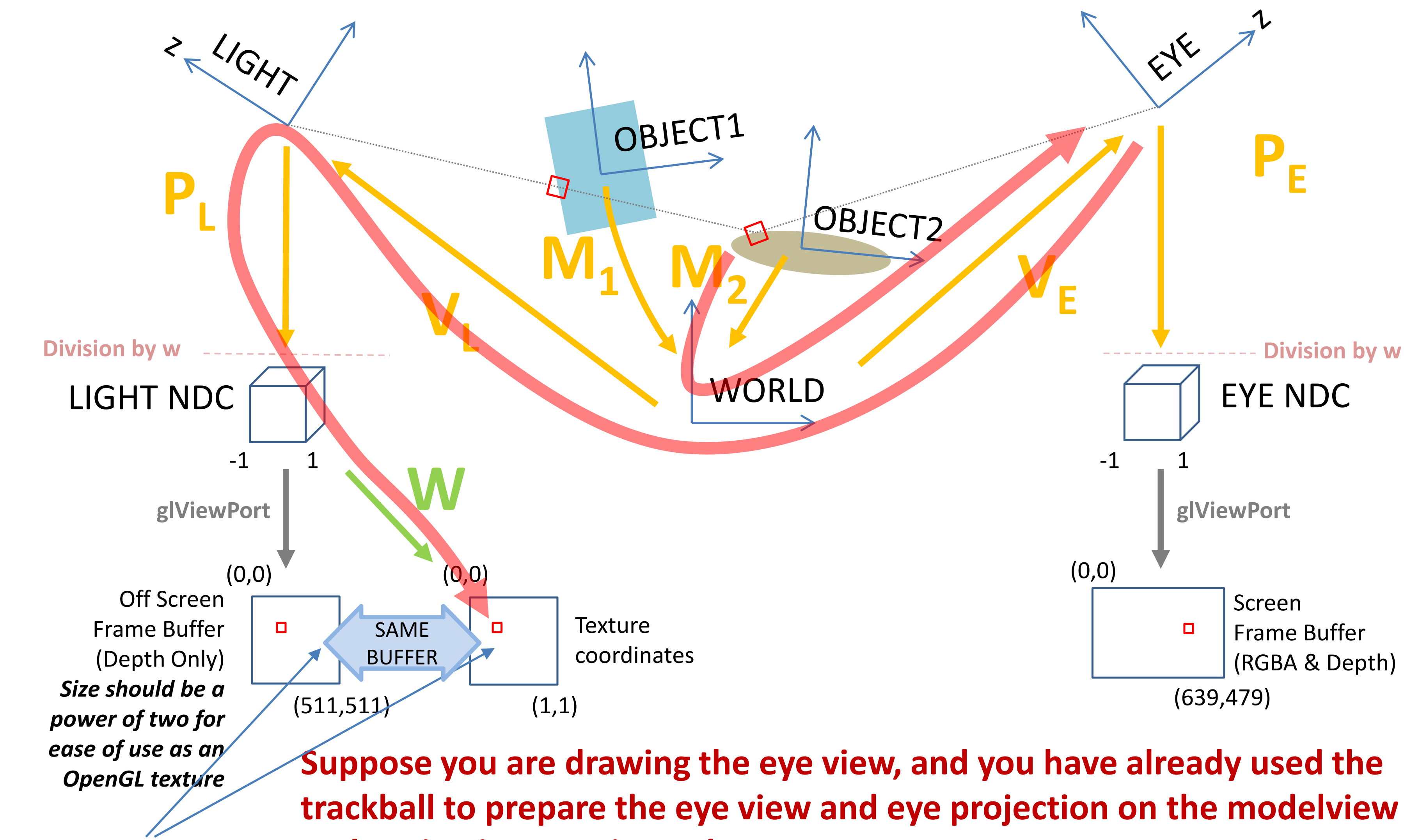
- **shadowMap** is the depth texture rendered from the light view
- Need to compute **pos** as the fragment position…
  - transformed to the light view coordinates,
  - projected into the light's normalized device coordinates NDC
  - Include divide through by w, *and* mapped to [0,1]x[0,1] through a windowing transformation as this is the range used for texture lookups, ***not the [-1,1]x[-1,1] of the NDC***.

```
float distanceFromLight = texture2D( shadowMap, pos.xy ).r
```

- Then compare **distanceFromLight** of the closest surface to the fragment you are currently shading (which is at distance **pos.z**). Note that you need to include some offset to avoid self shadowing!

Shadow Map lookup

- Matrix transformations to convert vertex locations to light NDC can be stored in a ***uniform*** matrix
  - Uniform is a keyword to denote values that are changed infrequently, and set as parameters to the vertex and or fragment programs before drawing a large number of primitives.
- The interpolated (varying) fragment location in NDC can then be used in the fragment program
  - Division by w is necessary before use.



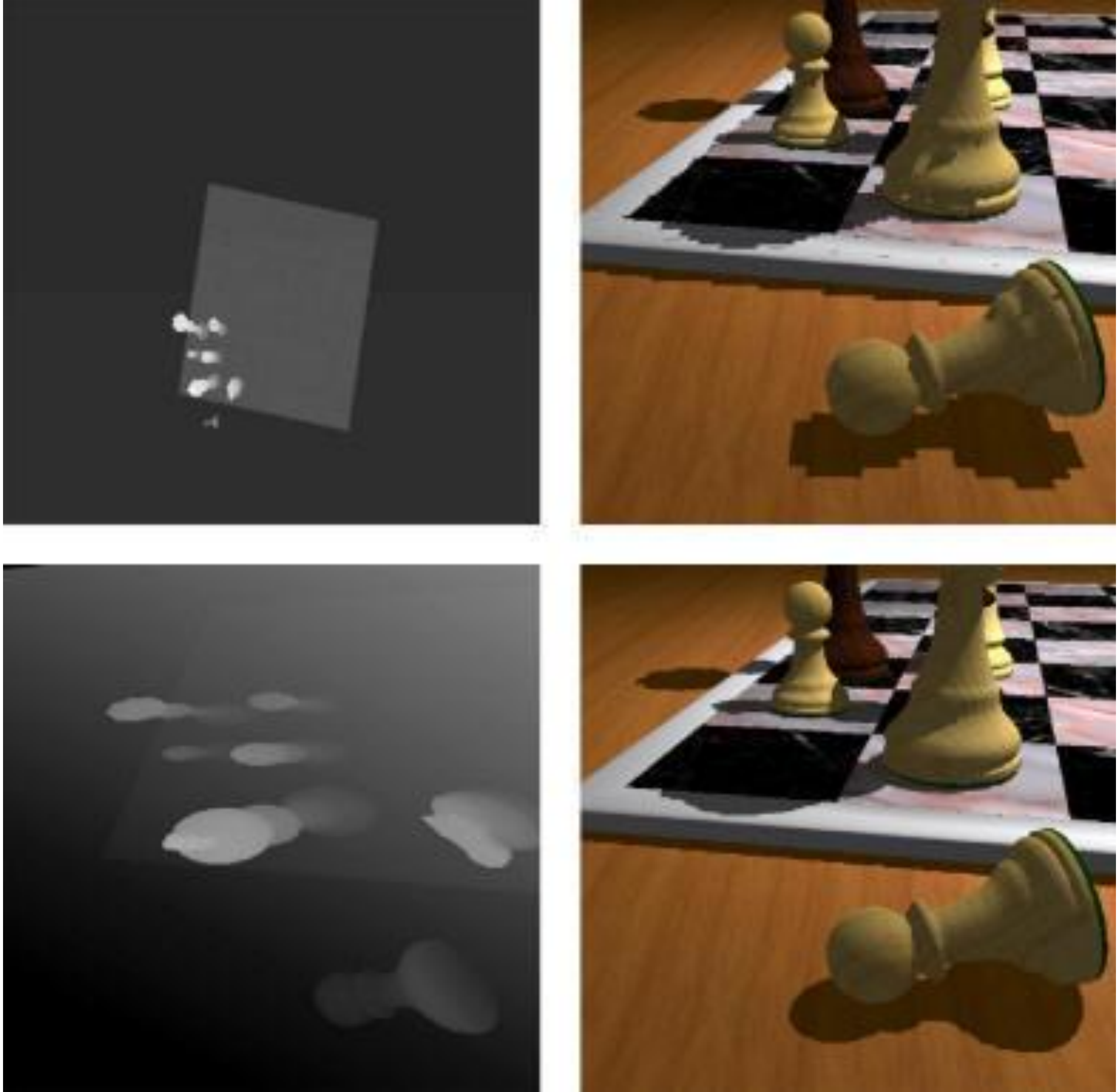
Same memory!
Just seen differently depending on if we are rendering to the buffer or accessing it as a texture.

Suppose you are drawing the eye view, and you have already used the trackball to prepare the eye view and eye projection on the modelview and projection matrix stacks. HOW DO YOU…

- **DRAW THE LIGHT COORDINATE FRAME?**
- **DRAW THE LIGHT FRUSTUM?**
- **CHECK THE SHADOWMAP FOR THE CURRENT FRAGEMENT?**

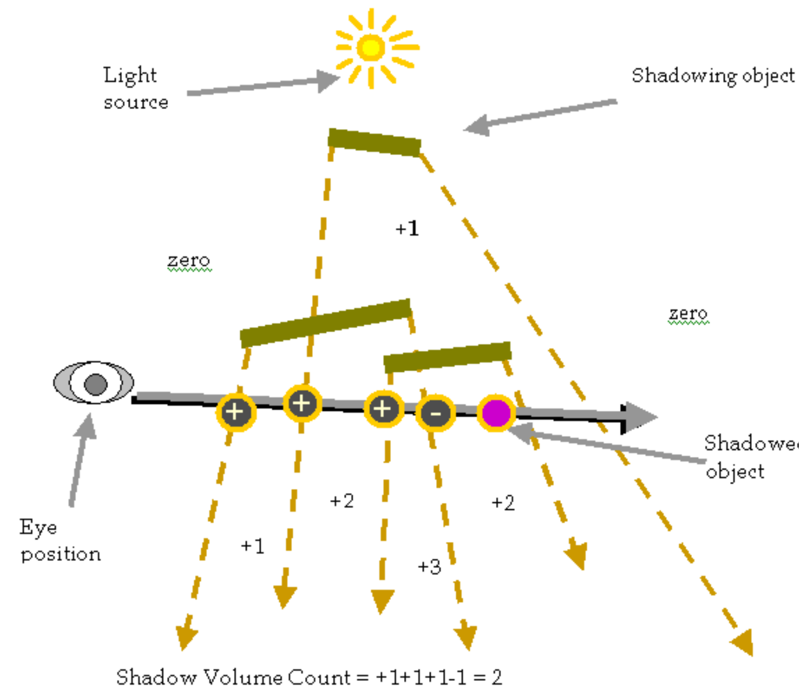
Shadow Maps

- Need to choose appropriate near / far / field of view for light
  - Demo
- Can use perspective shadow maps to improve accuracy in some cases
  - Aside: see GPU for your own satisfaction (not in the scope of the course) http://http.developer.nvidia.com/GPUGems/gpugems_ch14.html



Shadow Volumes

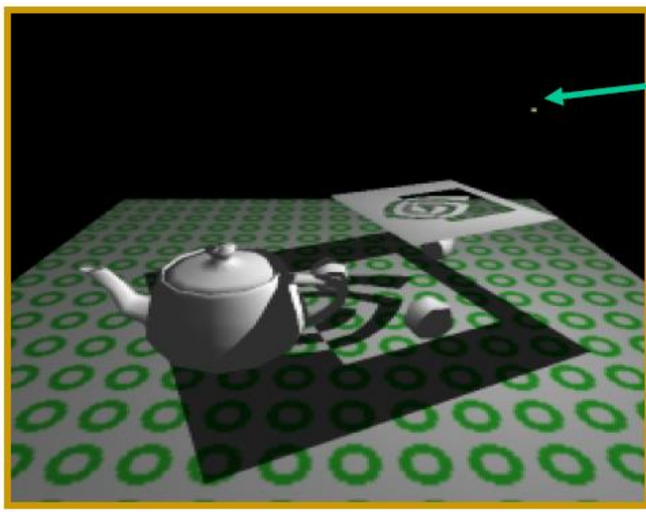
- Pixel-perfect shadows in screen space
- Use a special buffer, the ***stencil buffer***, and draw polygons associated with all silhouette edges



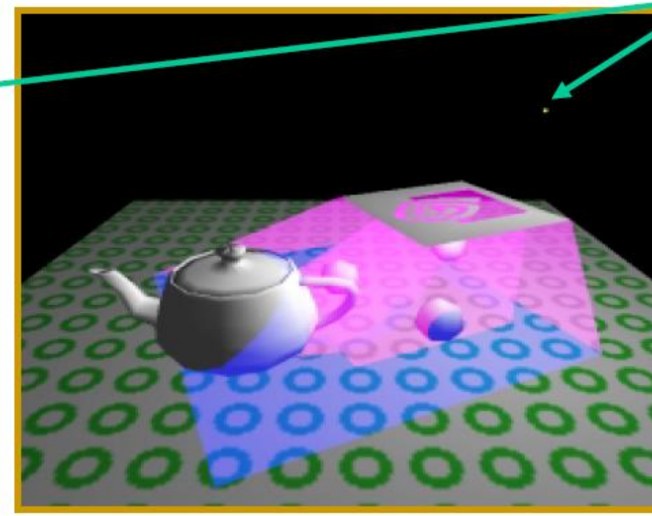
[Kuttuva]
[Everitt, Kilgard]

Visualizing Shadow Volumes

- Occluders and light source cast out a shadow volume
    - Objects within the volume should be shadowed



Scene with shadows from an NVIDIA logo casting a shadow volume

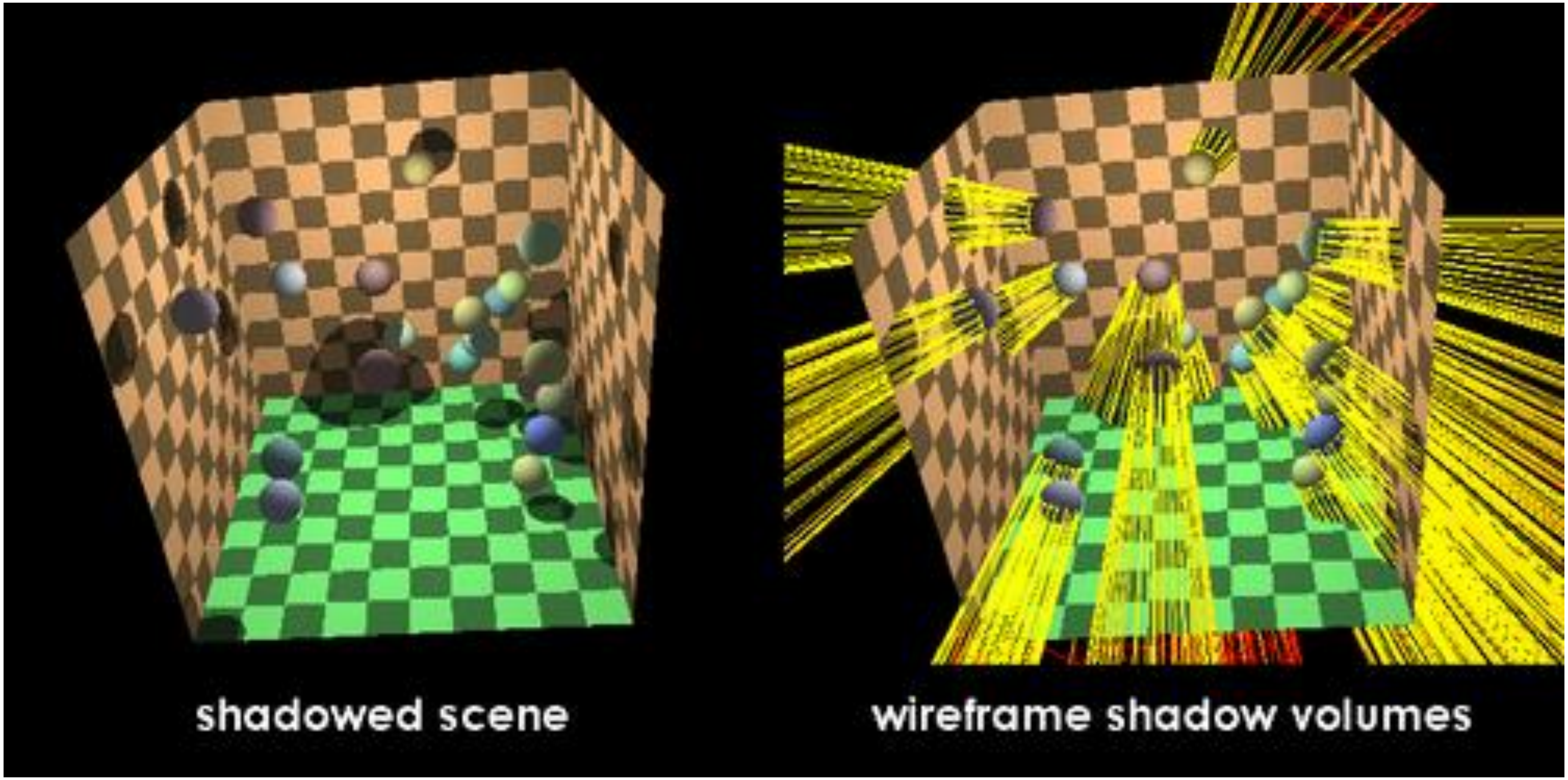


Visualization of the shadow volume

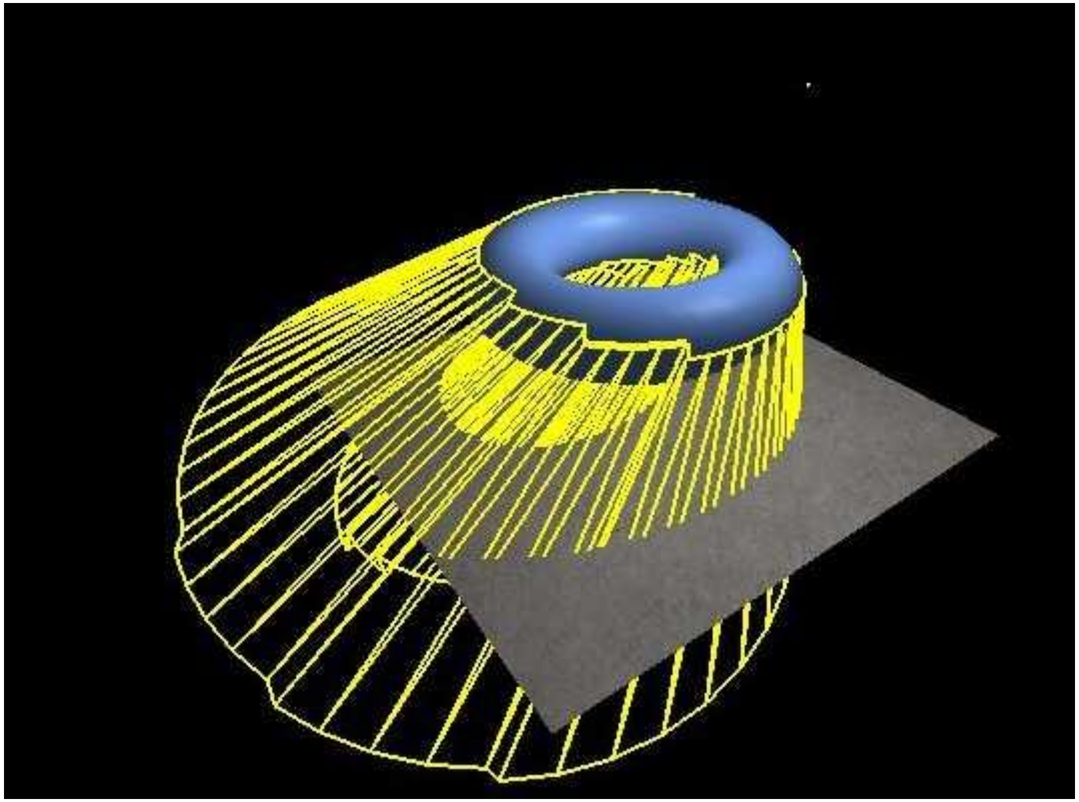
NVIDIA

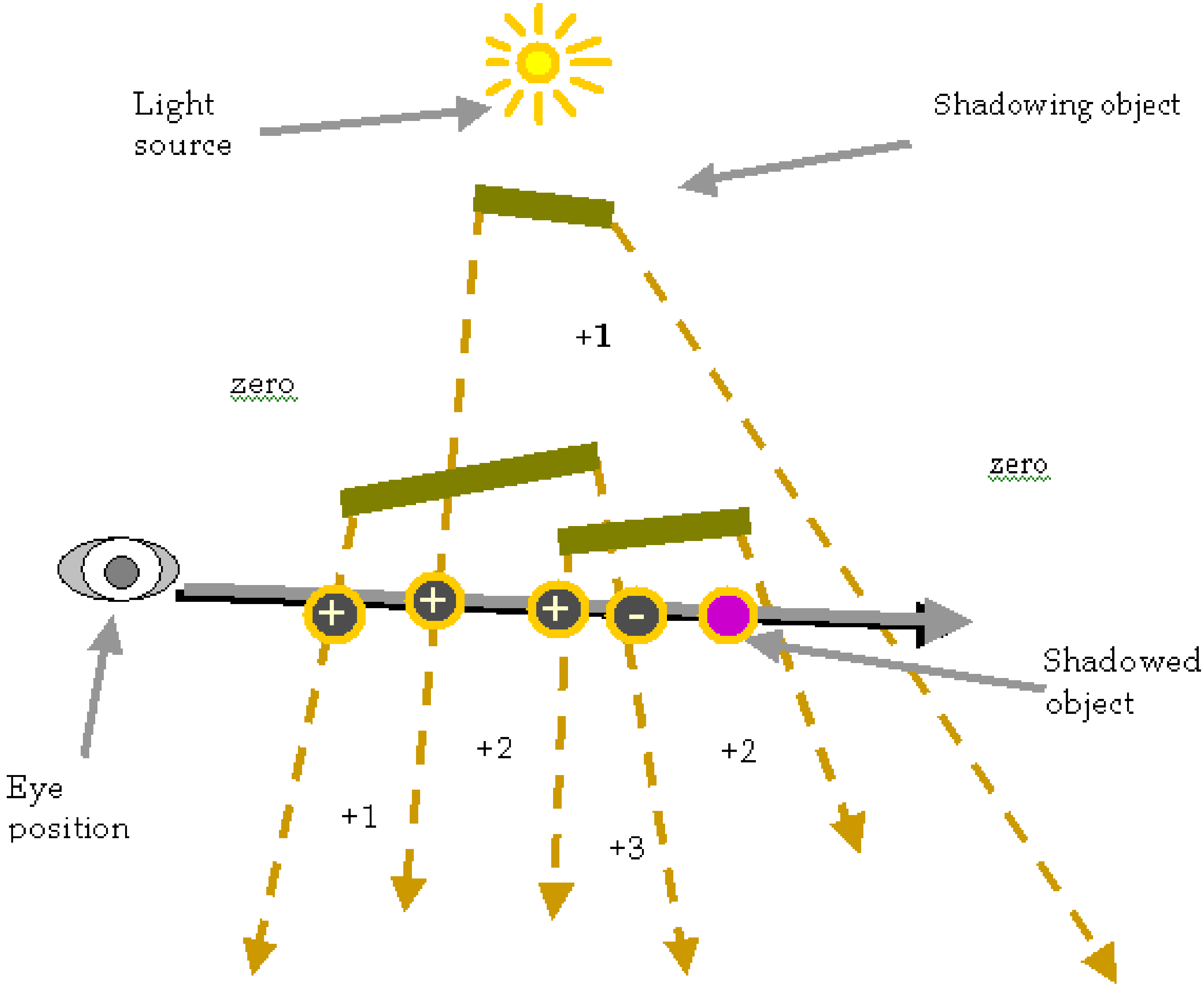
[From NVIDIA: Robust Stencil Shadow Volumes CEDEC 2001 Tokyo]

Shadow Volumes



Shadow Volumes





Shadow Volume Count = +1+1+1-1 = 2

Several Variations

- Depth pass description from Wikipedia:
  - Disable writes to the depth and color buffers.
  - Use back-face culling.
    - Set the stencil operation to increment on depth pass (only count shadows in front of the object).
    - Render the shadow volumes (because of culling, only their front faces are rendered).
  - Use front-face culling.
    - Set the stencil operation to decrement on depth pass.
    - Render the shadow volumes (only their back faces are rendered).
- See GPU Gems and NVIDIA developer zone for more information

Mirror reflection?

- Consider perfectly shiny surface
  - there isn't a highlight
  - instead there's a reflection of other objects
- How can we create a mirror in an object order rendering?
- "Glazed" material has mirror reflection and a diffuse component.

$$L = L_a + L_d + L_m$$