Intermediate Graphics & Animation Programming

GPR-300
Daniel S. Buckstein

Projective Texturing & Shadow Mapping
Week 4

License

- This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

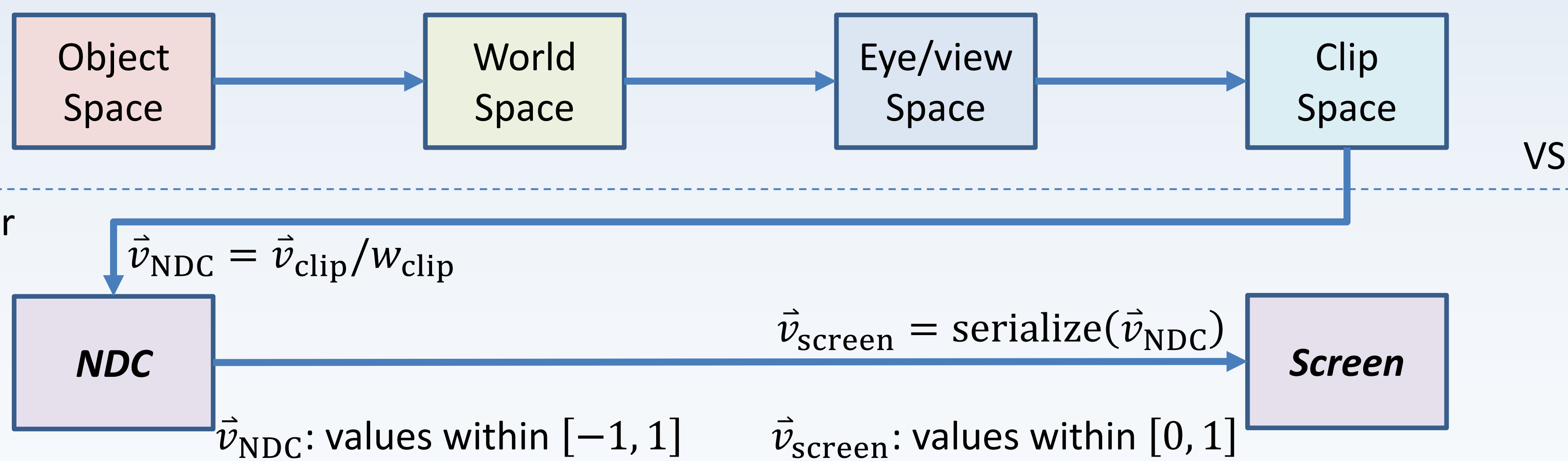
Shadow Mapping

- Review of vertex transformations
  - Clipping, review of 3D spaces
- Projective texturing
  - Yes, just like a PowerPoint shown on a projector
- Shadow mapping
  - The shadow pass
  - Applying the shadow… properly
- Intro to advanced algorithms

Review of Vertex Transformations

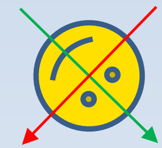
- Vertex transformation pipeline:
- The goal for the ***vertex shader*** is ***<u>clip space</u>***
- We'll call this the "*clipping pipeline*"

$$\vec{v}_{\text{object}} \times M = \vec{v}_{\text{world}} \times V = \vec{v}_{\text{eye}} \times P = \vec{v}_{\text{clip}}$$



Review of Vertex Transformations

- Model, view & projection matrices



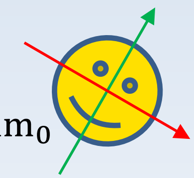
$M_{\mathrm{cam}_1} = {}^{\mathrm{world}}T_{\mathrm{cam}_1}$

$V_{\mathrm{cam}_1} = M_{\mathrm{cam}_1}^{-1}$

$VP_{\mathrm{cam}_1} = P_{\mathrm{cam}_1} \cdot V_{\mathrm{cam}_1}$

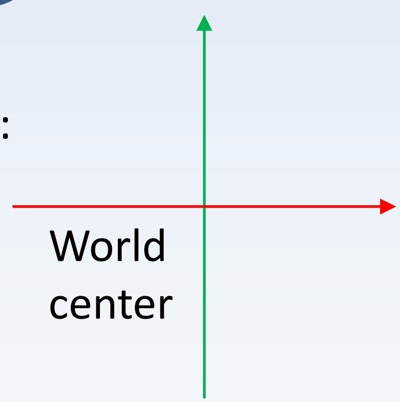
Eye/viewer/camera/observer/w.e.
relative to world

$M_{\mathrm{cam}_0} = {}^{\mathrm{world}}T_{\mathrm{cam}_0}$



View matrix: world relative to viewer:

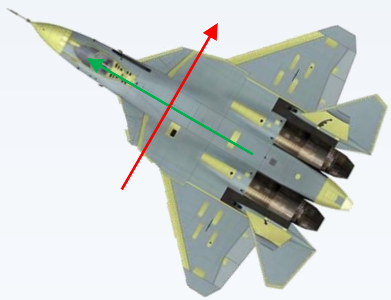
$V_{\mathrm{cam}_0} = M_{\mathrm{cam}_0}^{-1} = {}^{\mathrm{world}}T_{\mathrm{cam}_0}^{-1}$



World
center

#hypeplane ***Model matrix:***
relative to world

$M_{\mathrm{plane}} = {}^{\mathrm{world}}T_{\mathrm{plane}}$

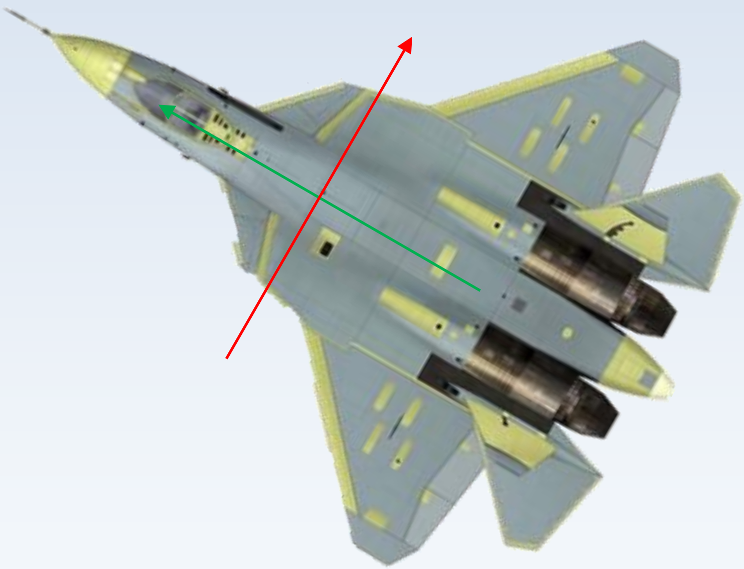


View-projection matrix
for this viewer:

$VP_{\mathrm{cam}_0} = P_{\mathrm{cam}_0} \cdot V_{\mathrm{cam}_0}$

Review of Vertex Transformations

- Model, view & projection matrices



#hypeplane ***Model matrix:***
relative to world

$$M_{\text{plane}} = {}^{\text{world}}T_{\text{plane}}$$

$$MVP_{\text{plane,cam}_0} = \left(VP_{\text{cam}_0}\right) \cdot M_{\text{plane}}$$
$$= \left(P_{\text{cam}_0} \cdot V_{\text{cam}_0}\right) \cdot M_{\text{plane}}$$

$$MVP_{\text{plane,cam}_1} = \left(VP_{\text{cam}_1}\right) \cdot M_{\text{plane}}$$

$$MVP_{\text{plane,cam}_2} = \left(VP_{\text{cam}_2}\right) \cdot M_{\text{plane}}$$

…etc.

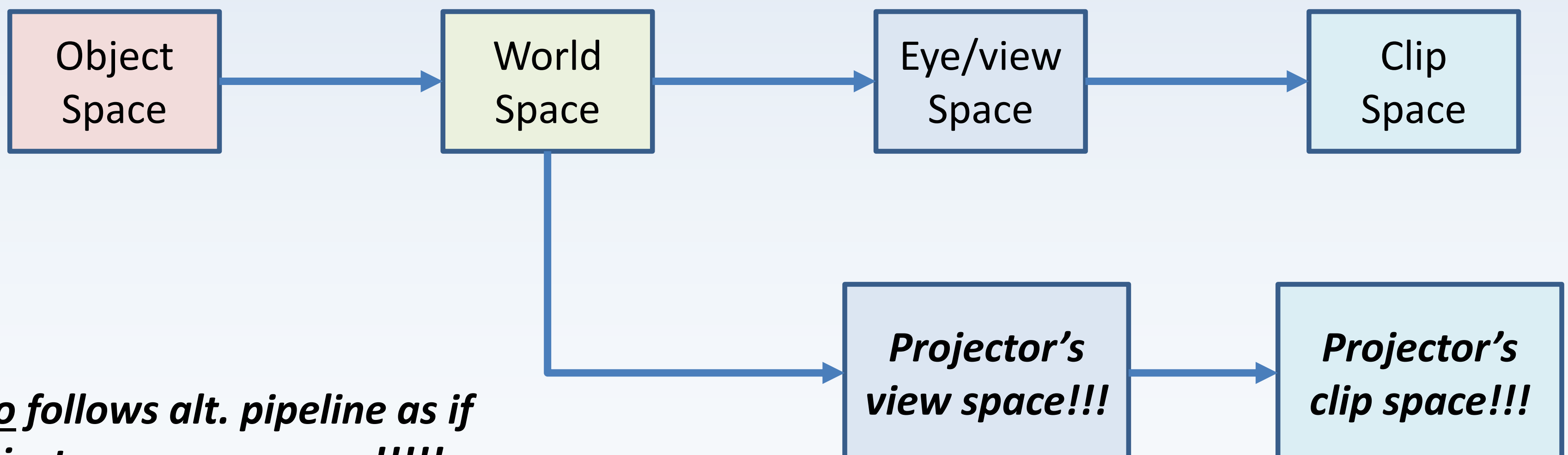
Projective Texturing

- The concept:
- Drawing an object, we will see it on-screen
- The question: ***does the projector also see it?***
- Projector treated as “alternate camera”
- If projector can see the object we are drawing, we apply a texture to it ☺

Projective Texturing

- Vertex being processed in VS follows *clipping pipeline* as normal
  - …because it has to

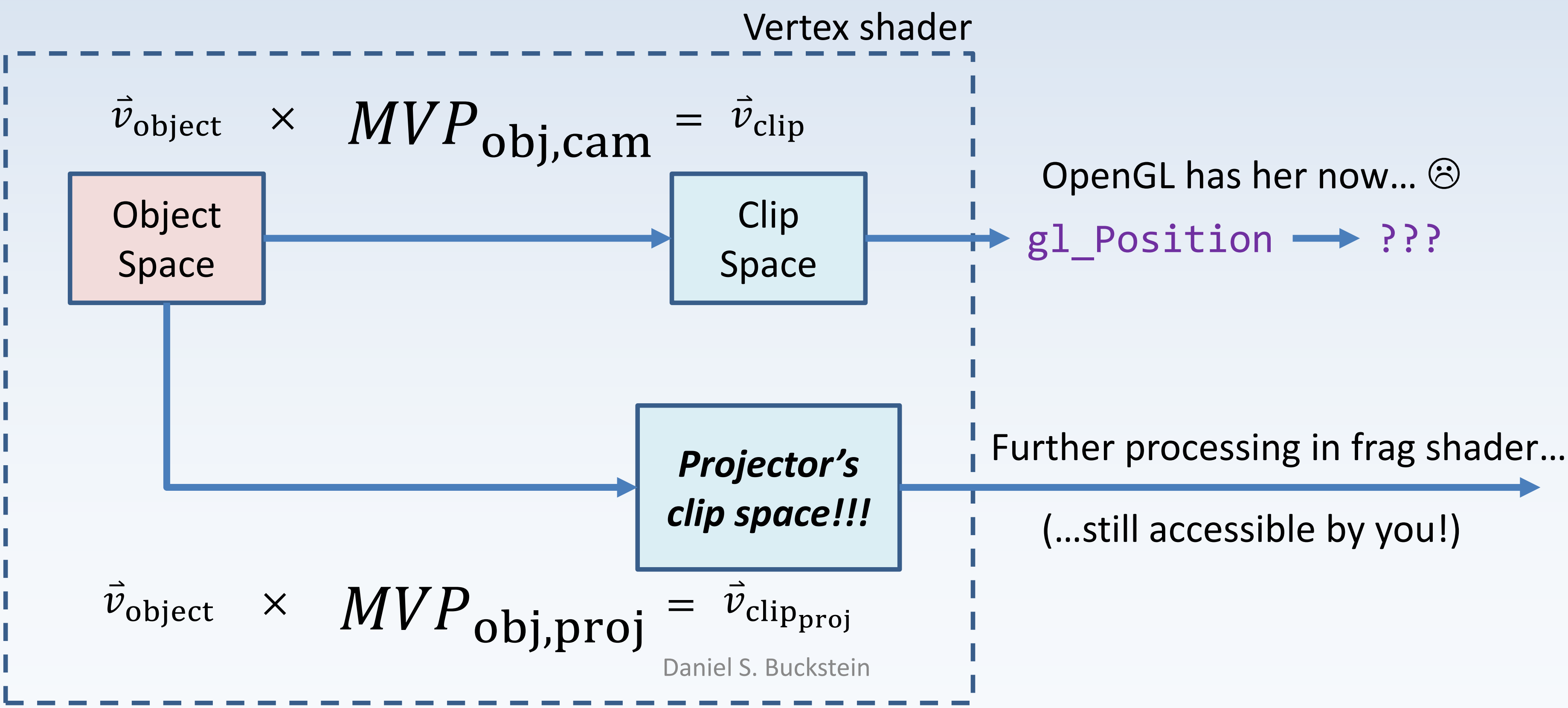
$$\vec{v}_{\text{object}} \times M = \vec{v}_{\text{world}} \times V = \vec{v}_{\text{eye}} \times P = \vec{v}_{\text{clip}}$$



<u>Also</u> follows alt. pipeline as if projector was a camera!!!!!

Projective Texturing

- We have the complete clipping pipeline…
- …and an alternative pipeline representing…?



Daniel S. Buckstein

Projective Texturing

- Position in projector's clip space is **not part of *graphics pipeline***
- It is our own variable… what happens next???

Fragment shader

$$\vec{v}_{\mathrm{clip}_{\mathrm{proj}}} \xrightarrow{/w_{\mathrm{clip}_{\mathrm{proj}}}} \vec{v}_{\mathrm{NDC}_{\mathrm{proj}}} \xrightarrow{\text{serialize}} \vec{v}_{\mathrm{screen}_{\mathrm{proj}}}$$

Non-"clipped" values: $[-1, 1]$

Non-"clipped" values: $\boldsymbol{[0, 1]}$

Projective Texturing

- Pro optimization tip: serializing per-fragment is not time-friendly
- Use the power of matrices to do this in the vertex shader

Fragment shader

$$\longrightarrow \boxed{\vec{v}_{\mathrm{clip}_{\mathrm{proj}}}} \xrightarrow[/w_{\mathrm{clip}_{\mathrm{proj}}}]{} \boxed{\vec{v}_{\mathrm{NDC}_{\mathrm{proj}}}} \xrightarrow[\text{serialize}]{} \boxed{\vec{v}_{\mathrm{screen}_{\mathrm{proj}}}}$$

Projective Texturing

- Concatenate a matrix *after projection* that performs “perspectively-correct” serialization

$$\vec{v}'_{\mathrm{clip}_{\mathrm{proj}}} = \begin{bmatrix} 0.5 & & & 0.5 \\ & 0.5 & & 0.5 \\ & & 0.5 & 0.5 \\ & & & 1 \end{bmatrix} \vec{v}_{\mathrm{clip}_{\mathrm{proj}}}$$

“Bias matrix” B

$$\begin{aligned} \vec{v}'_{\mathrm{clip}_{\mathrm{proj}}} &= (MVPB)_{\mathrm{obj,proj}} \times \vec{v}_{\mathrm{obj}} \\ &= (B \times P_{\mathrm{proj}} \times V_{\mathrm{proj}} \times M_{\mathrm{obj}}) \times \vec{v}_{\mathrm{obj}} \end{aligned}$$

Projective Texturing

- Projective texturing VS (120):

```
attribute vec4 vertex; // ...and other attribs
uniform mat4 mvp_main;          // main pipeline
uniform mat4 mvp_proj_bias;     // alt. pipeline
varying vec4 projClip;
void main() {
    gl_Position = mvp_main * vertex; //required
    projClip = mvp_proj_bias * vertex;
// ...anything else you need to do in your VS
}
```

Projective Texturing

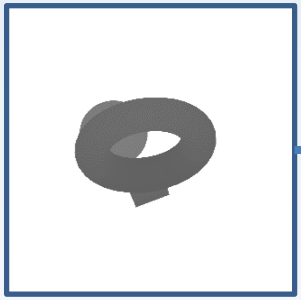
- Projective texturing FS (120):

```
varying vec4 projClip;
uniform sampler2D projTexture;
void main() {
    // pre-serialized perspective divide!!! :D
    vec4 projScreen = projClip / projClip.w;
    // assign fragment colour from texture!
    gl_FragColor = texture2D (projTexture,
        projScreen.xy);
}
```

Shadow Mapping

- ***Shadow mapping*** is an application of projective texturing
- Requires one additional pass and some additional processing in fragment shader

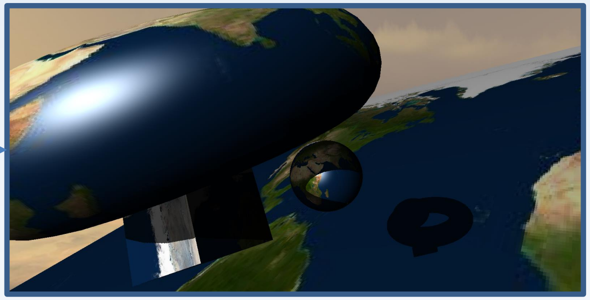
Pass 1: Acquire ***shadow map***



→Render scene from ***light's point of view***…

…with a twist

Pass 2: Draw scene, determine shadows!



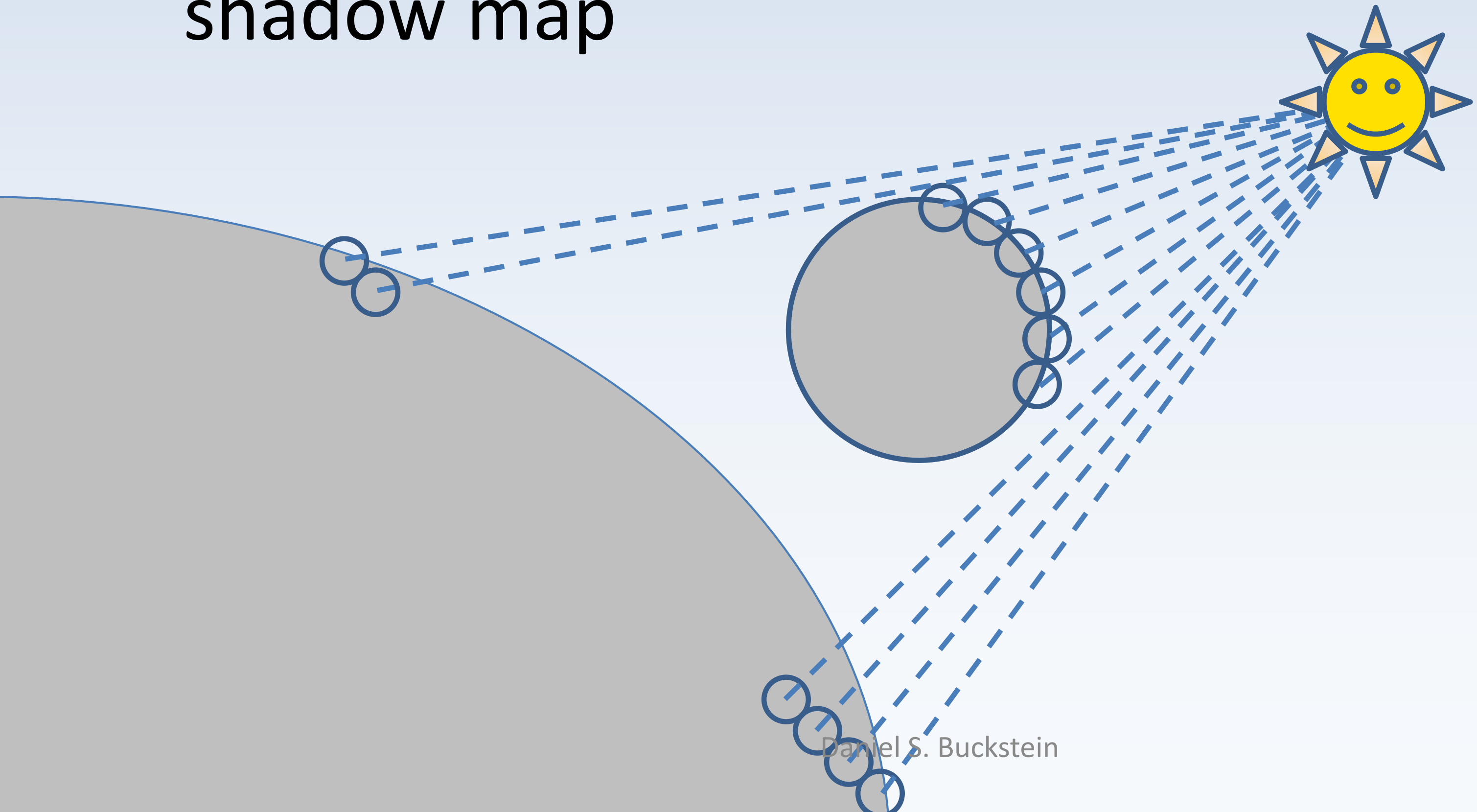
→Render scene using projective texturing
→Instead of projecting a texture, compute shadows using shadow map

Shadow Mapping

- First pass: acquire "***shadow map***": scene viewed from ***light's point of view***
- It's a standard render pass… but using the light as if it were a camera!!!
- Resulting depth map is called "shadow map"☺
- ***Food for thought***: What does the *vertex shader* for this pass look like???
- What does the *fragment shader* look like???

Shadow Mapping

- First pass: render scene from ***light's*** POV
- Resulting depth map (values [0, 1]) is the shadow map



Shadow Mapping

- Second pass: render scene from camera's POV
- Vertex shader: same as *projective texturing*

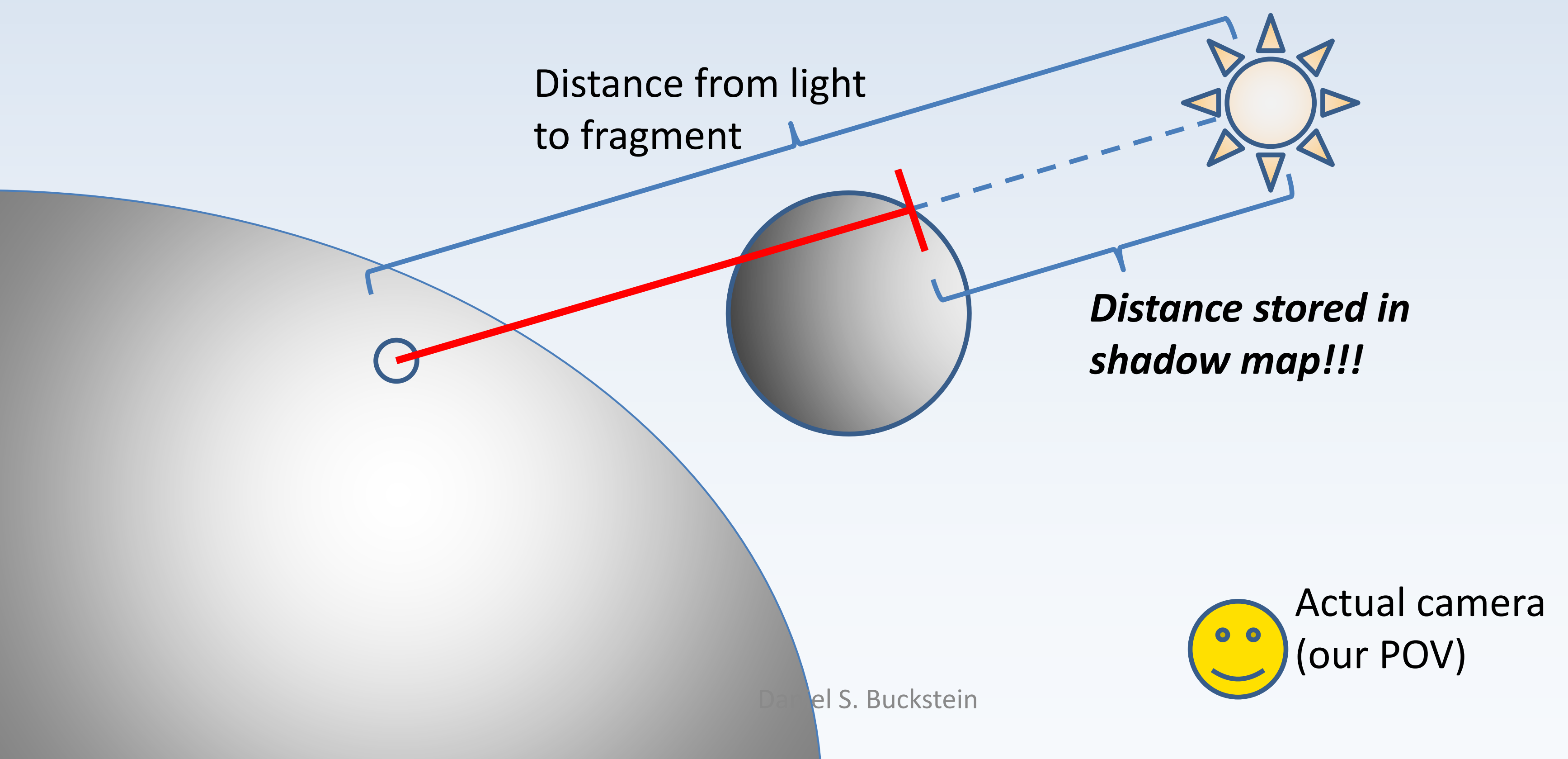
Vertex shader

$$\vec{v}_{\text{object}} \times MVP_{\text{obj,cam}} = \vec{v}_{\text{clip}}$$

$$\vec{v}_{\text{object}} \times MVPB_{\text{obj,light}} = \vec{v}'_{\text{clip}_{\text{light}}}$$

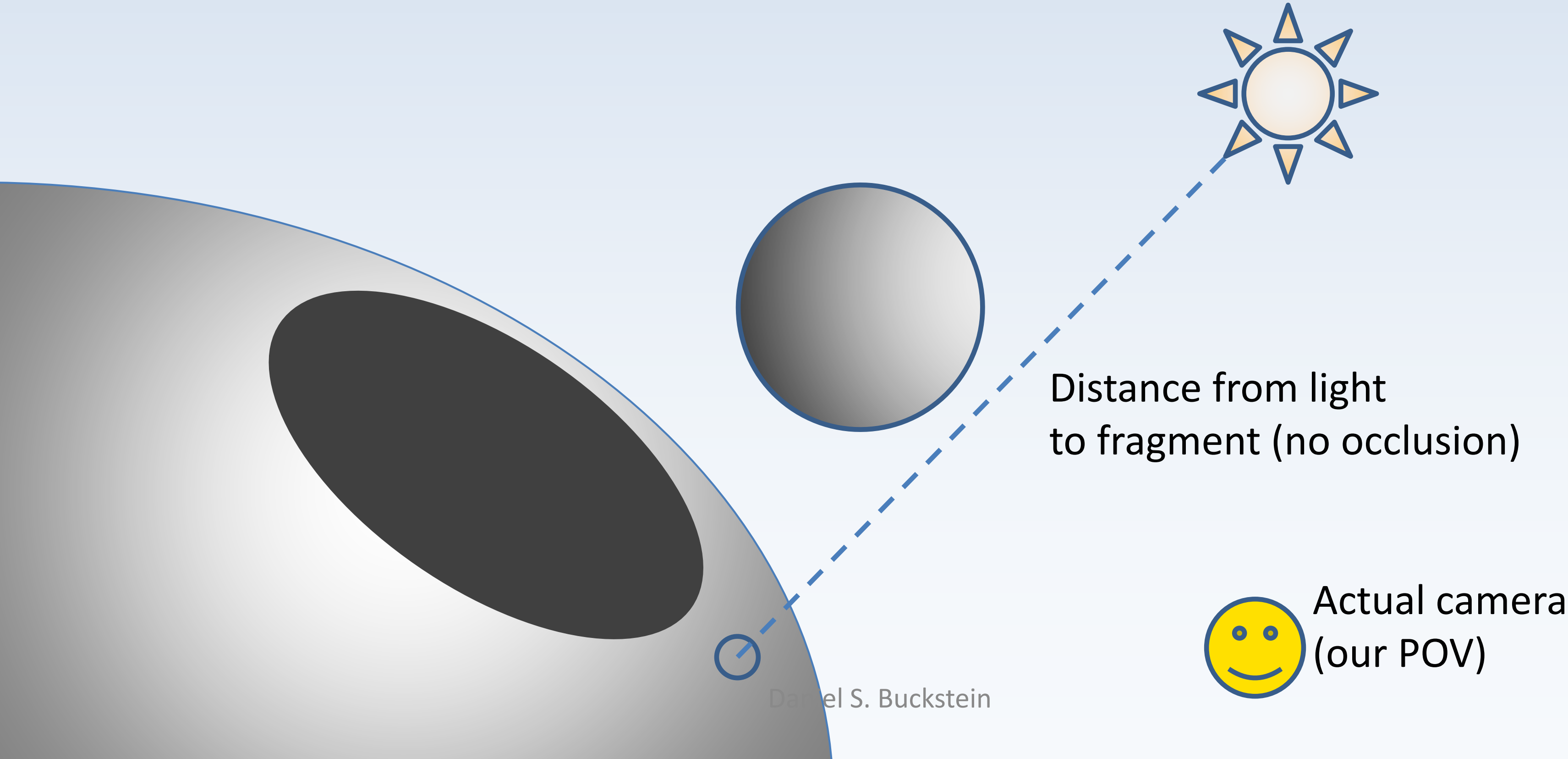
Shadow Mapping

- Second pass: render scene from camera's POV
- Fragment shader: a tiny bit different



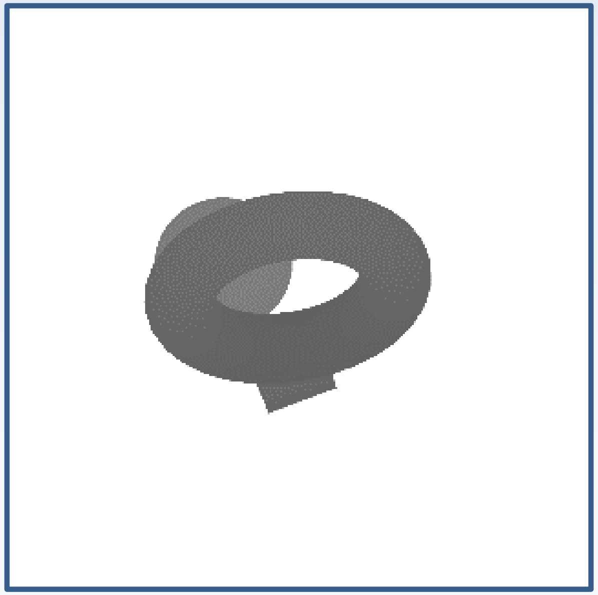
Shadow Mapping

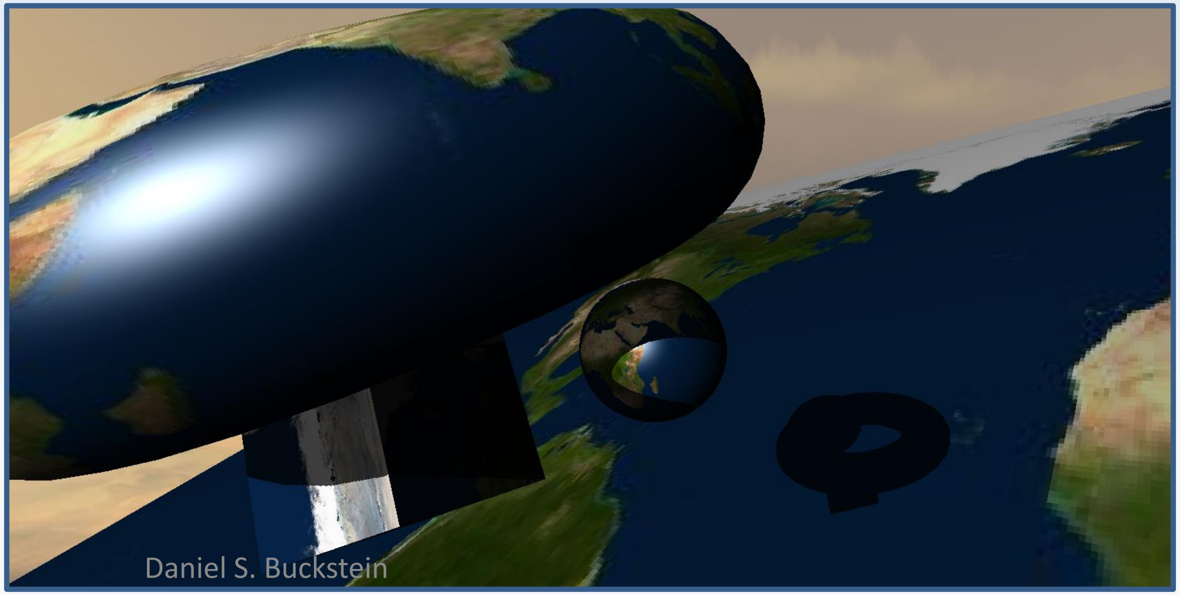
- Second pass: render scene from camera's POV
- Fragment shader: a tiny bit different



Shadow Mapping

- Shadow map is used for comparison of a fragment's distance from the light!
- If the distance is greater than that stored in the shadow map… ***fragment is in shadow!*** ☺





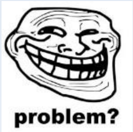
Shadow Mapping

- Shadow mapping FS (120): start with projective texturing FS…

```
uniform sampler2D projTexture;
uniform sampler2D shadowMap;
...
    gl_FragColor = texture2D (projTexture,
        projScreen.xy);
    float shadowSample = texture2D (shadowMap,
        projScreen.xy).r;
    bool fragIsShadowed =
        ( projScreen.z > shadowSample );
```

Shadow Mapping: Improvements

- ***#1) Integrate with lighting and shading***:
- Scale down diffuse and/or specular effect if the fragment is under shadow

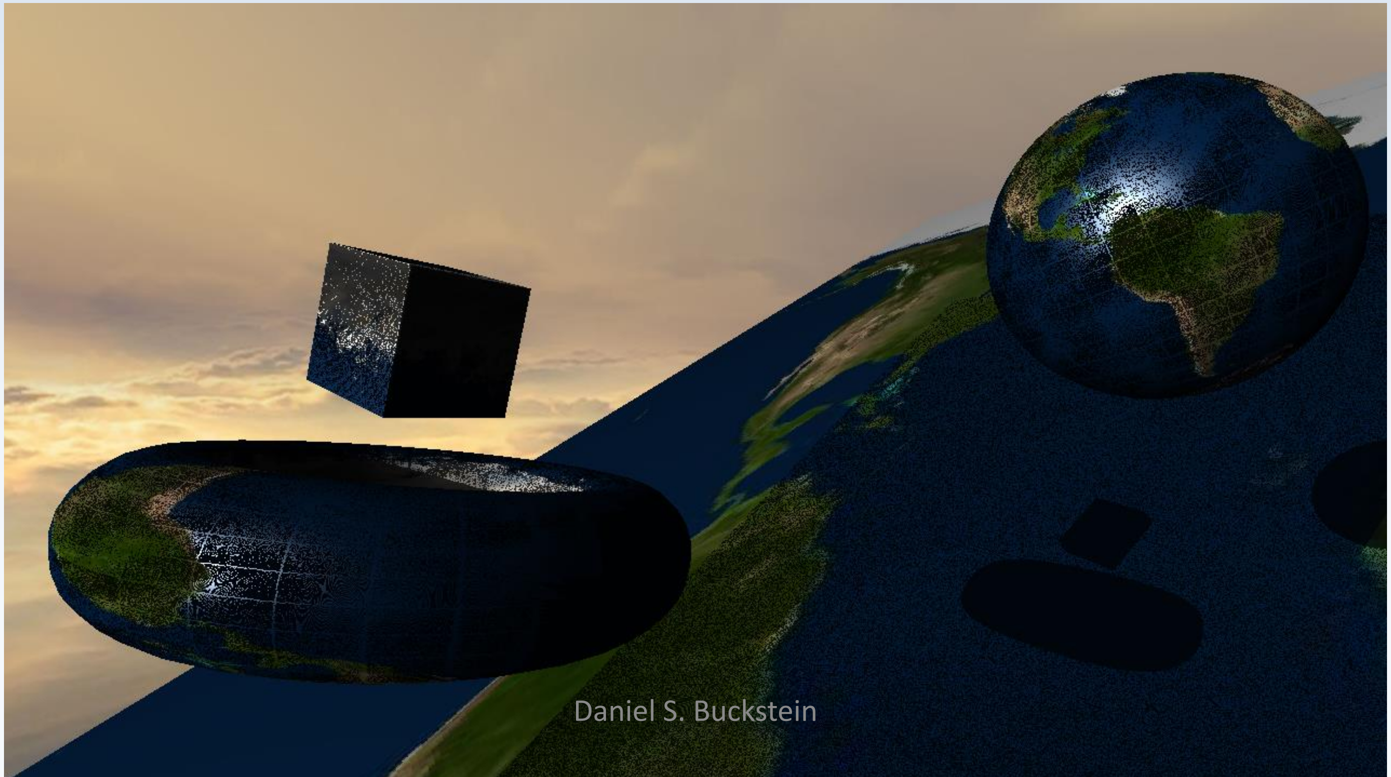



```
if (fragIsShadowed) {
    kd *= 0.2; // scale diffuse
}
// proceed with shading
```

Shadow Mapping: Improvements

- ***#2) Prevent Z-fighting***:

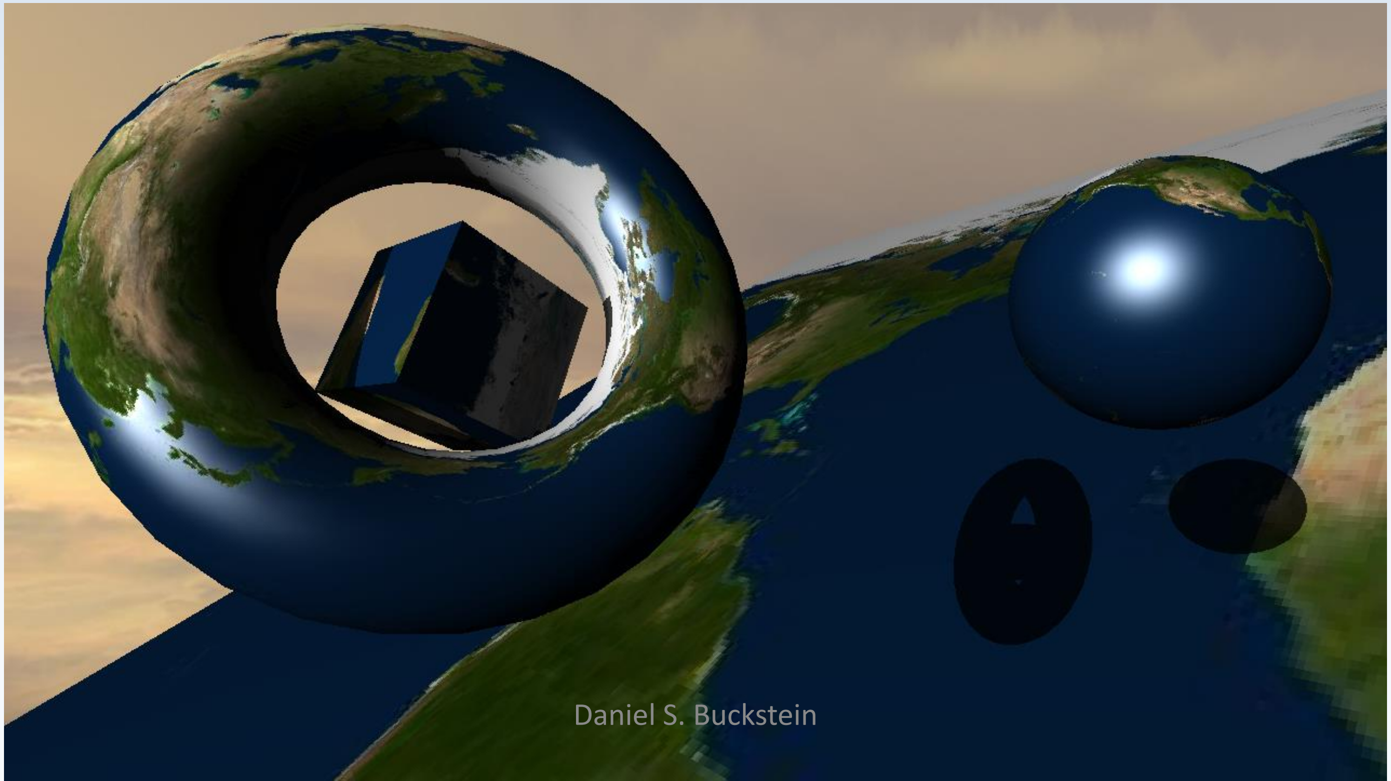
```
bool fragIsShadowed =
    ( projScreen.z > (shadowSample) );
```



Shadow Mapping: Improvements

- ***#2) Prevent Z-fighting***: add a *tiny* offset/bias

```
bool fragIsShadowed =
    ( projScreen.z > (shadowSample + 0.0025) );
```

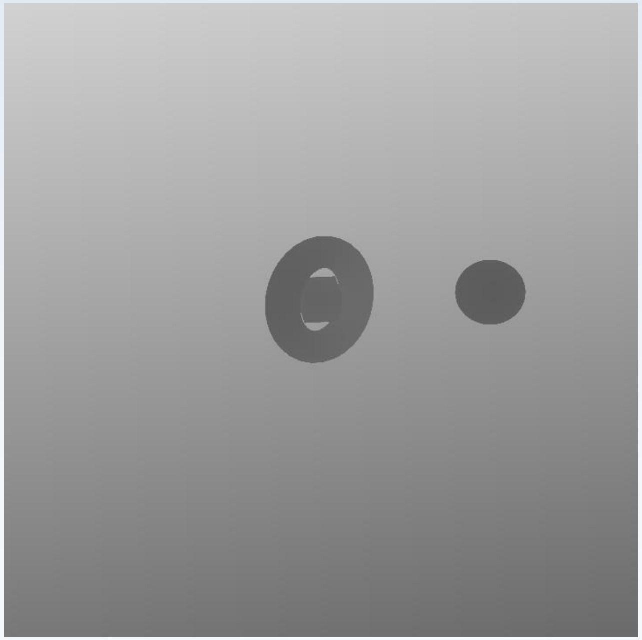


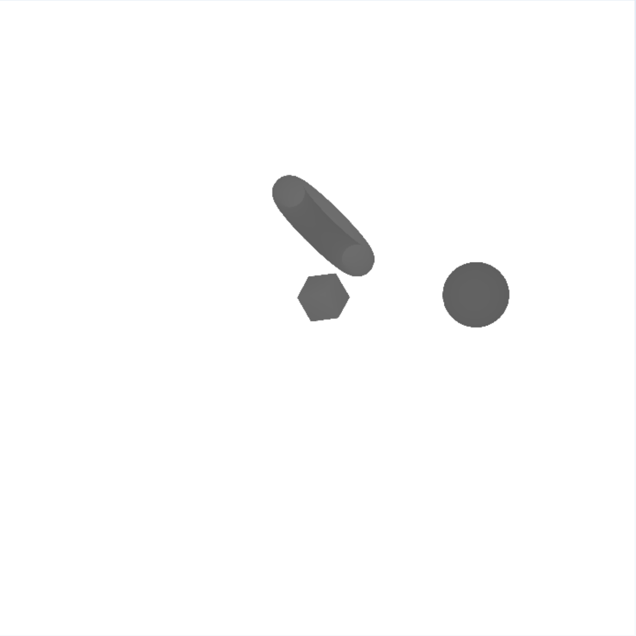
Shadow Mapping: Improvements

- ***#3) Front-face culling for shadow pass***:

```
// SHADOW PASS (before scene)
// regular back-face culling
DrawSceneObjects();
```

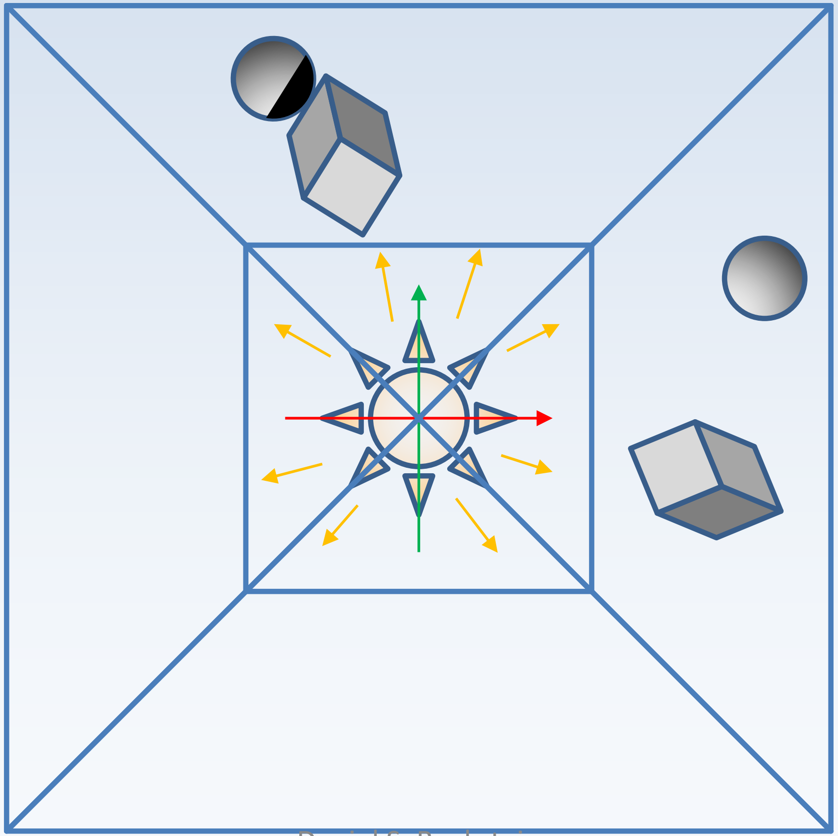
```
// SHADOW PASS (before scene)
glCullFace(GL_FRONT);
DrawSceneObjects();
glCullFace(GL_BACK);
```





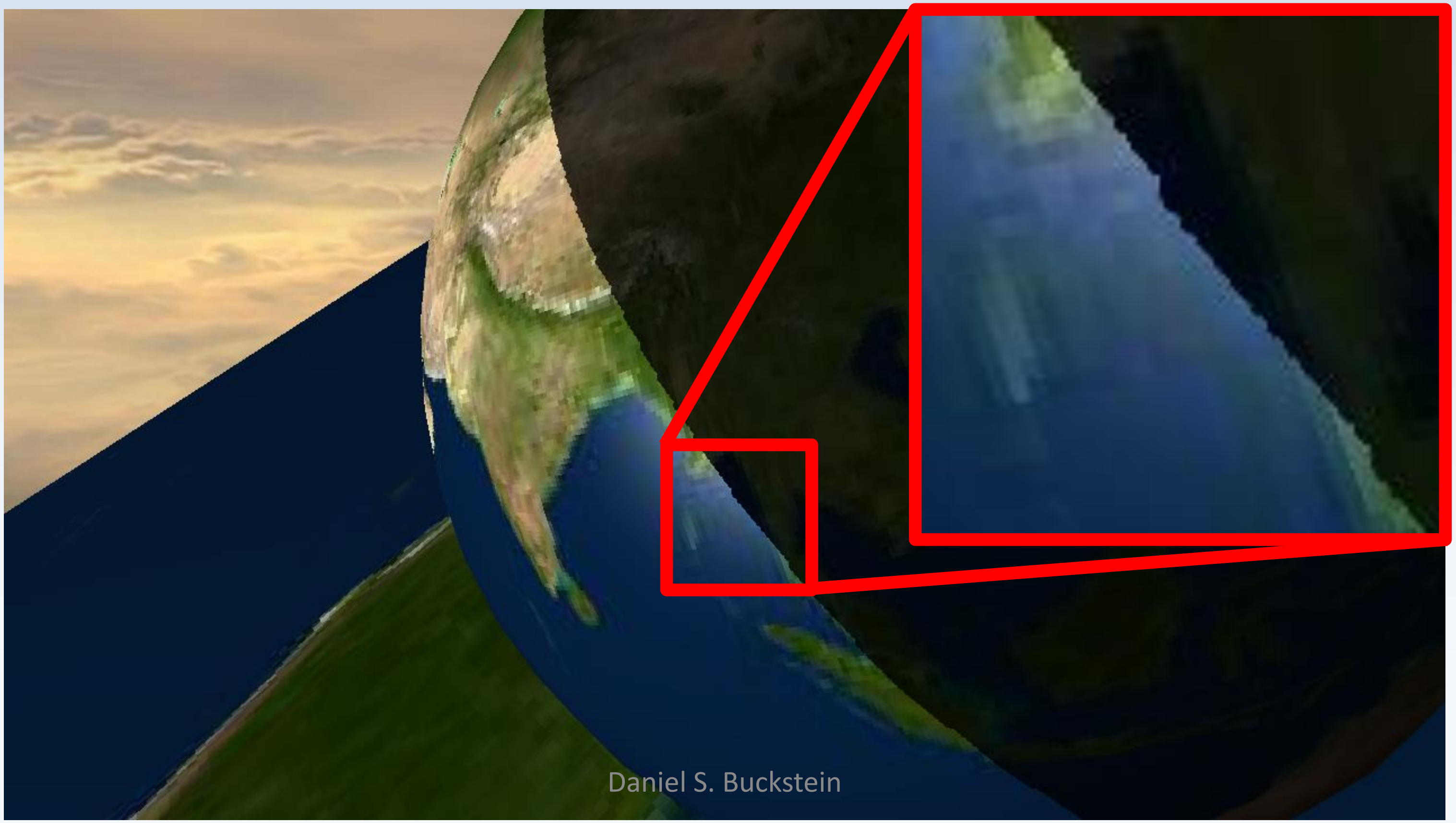
Intro to Advanced Algorithms

- How do we handle omni-directional lights???



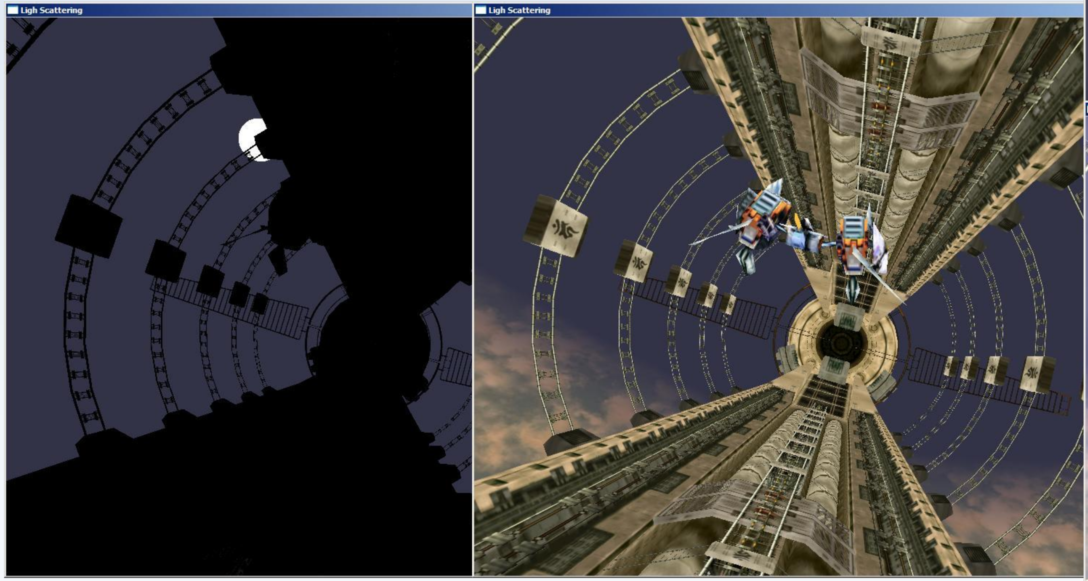
Intro to Advanced Algorithms

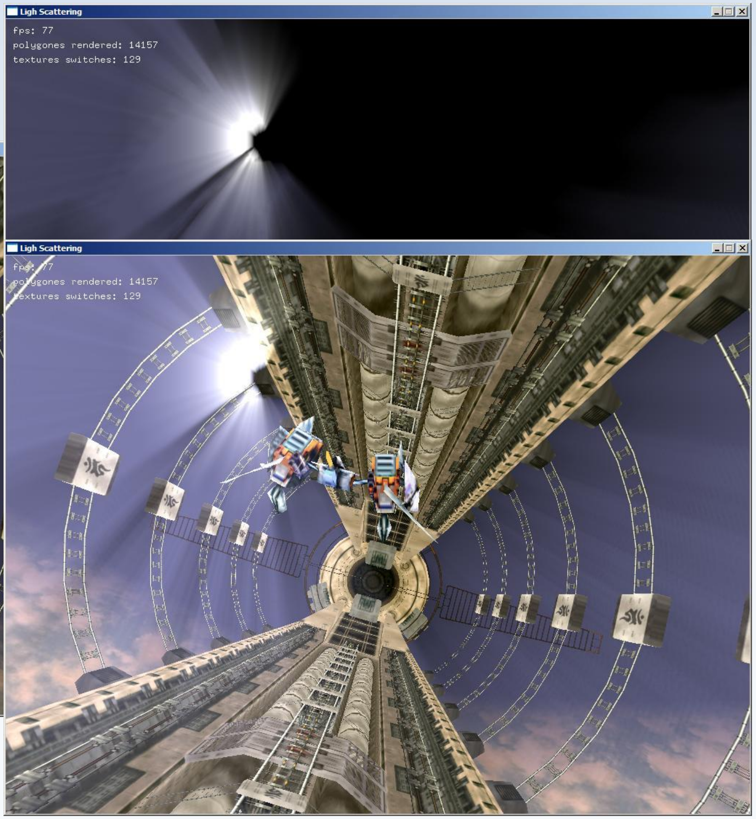
- Common problem with all shadow mapping shadows:



Intro to Advanced Algorithms

- God rays… behold!!! :o
- *Very basic* 3-pass ray-tracing algorithm
- Read the tutorial ☺





http://fabiensanglard.net/lightScattering/

Intro to Advanced Algorithms

- Omni-directional lights: create multiple shadow maps… inefficient for real-time!
- What if you render shadows at runtime and bake them into texture maps? ☺
- Smooth shadows: PCSS and others
- God rays... so pretty… combine with bloom?
- Lots of stuff to do with shadows!
- See the books for complete details!

The end.

- Questions? Comments? Concerns?