CS482: Interactive Computer Graphics

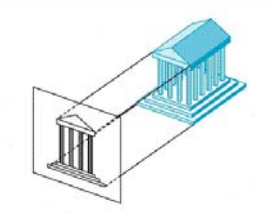
Min H. Kim
KAIST School of Computing

Chapter 15

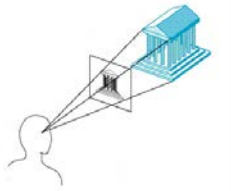
CAMERA MAPPING

Camera projection

- Perspective projection
- Orthogonal projection



Orhtogonal projection

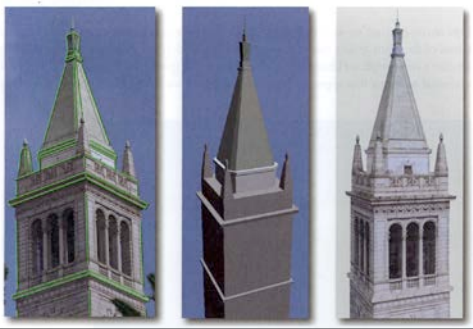


Perspective projection

http://jcsites.juniata.edu/faculty/rhodes/graphics/viewing.htm

Projector texture mapping

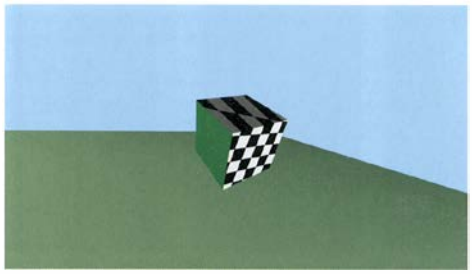
- There are times when we wish to glue our texture onto our triangles using a *projector* model, instead of the affine gluing model.
- For example, we may wish to simulate a slide projector illuminating some triangles in space.



Projector texture mapping

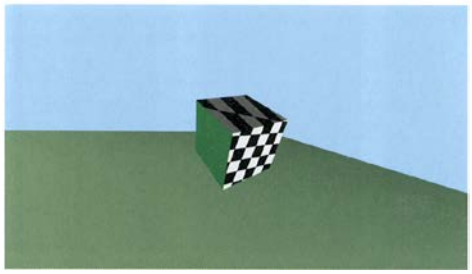
- The slide projector is modeled using 4 by 4, modelview and projection matrices, M_s and P_s

$$\begin{bmatrix} x_t w_t \\ y_t w_t \\ - \\ w_t \end{bmatrix} = P_s M_s \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$



Projector texture mapping

- With the texture coordinates defined as $x_t = \frac{x_t w_t}{w_t}$ and $y_t = \frac{y_t w_t}{w_t}$
- To color a point on a triangle with object coordinates $[x_o, y_o, z_o, 1]^t$, we fetch the texture data stored at location $[x_t, y_t]^t$



Projector texture mapping

- The three quantities $x_t w_t$, $y_t w_t$ and w_t are all affine functions of (x_o, y_o, z_o). Thus these quantities will be properly interpolated over a triangle when implemented as varying variables.
- In the fragment shader, we need to divide by w_t to obtain the actual texture coordinates.
- When doing projector texture mapping, we do not need to pass any texture coordinates as attribute variables to our vertex shader.

Projector texture mapping

- We simply use the object coordinates already available to us.
- We do need to pass in, using uniform variables, the necessary projector matrices.

Projector texture mapping

- Projector vertex shader

```
#version 130

uniform mat4 uModelViewMatrix;
uniform mat4 uProjMatrix;

uniform mat4 uSProjMatrix;
uniform mat4 uSModelViewMatrix;

in vec4 aVertex;
out vec4 aTexCoord;

void main(){
  vTexCoord = uSProjMatrix * uSModelViewMatrix * aVertex;
  gl_Position = uProjMatrix * uModelViewMatrix * aVertex;
}
```


Projector texture mapping

- Projector fragment shader

```
#version 130

uniform sampler2D vTexUnit0;

in vec4 aTexCoord;
out vec4 fragColor;

void main(){
  vec2 tex2;
  tex2.x = vTexCoord.x/vTexCoord.w;
  tex2.y = vTexCoord.y/vTexCoord.w;
  vec4 texColor0 = texture2D(vTexUnit0, tex2);
  fragCoor = texColor0;
}
```

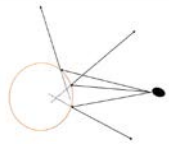
Projector texture mapping

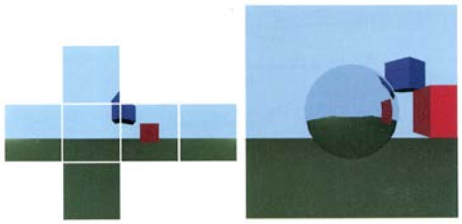
- Conveniently, OpenGL even gives us a special call texture2DProj(vTexUnit0, pTexCoord), that actually does the divide for us.
- Inconveniently, when designing our slide projector matrix uSProjMatrix, we have to deal with the fact that the canonical texture image domain in OpenGL is the unit square, whose lower left and upper right corners have coordinates $[0,0]^t$ and $[1,1]^t$ used for the display window.

Multipass Rendering

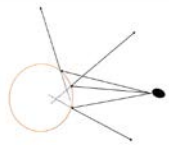
- More interesting rendering effects can be obtained using multiple rendering passes over the geometry in the scene.
- In this approach, the results of all but the final pass are stored offline and not drawn to the screen.
- To do this, the data is rendered into something called, a FrameBufferObject, or FBO.
- After rendering, the FBO data is then loaded as a texture, and thus can be used as input data in the next rendering pass.

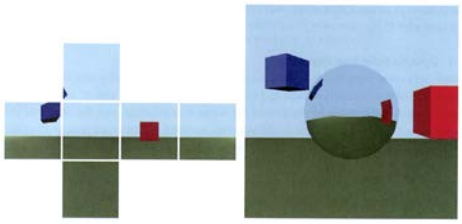
Multipass





Multipass



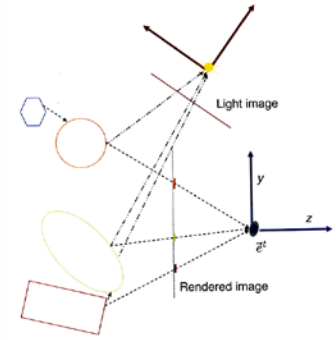


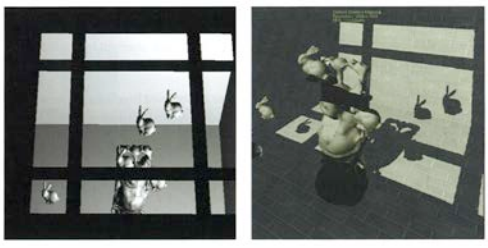
Chapter 15

SHADOW MAPPING

Shadow mapping

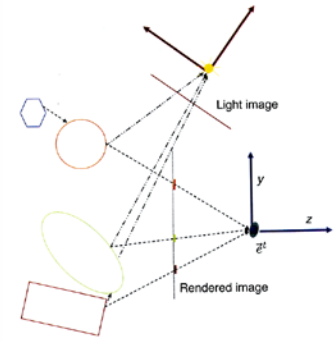
- The idea is to first create and store a z-buffered image from the point of view of the light, and then compare what we see in our view to what the light saw it its view.

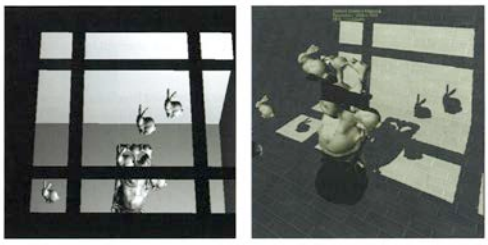




Shadow mapping

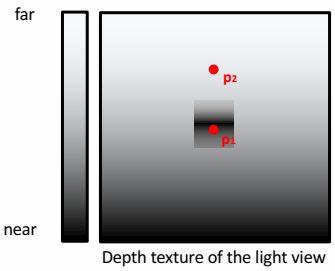
- If a point observed by the eye is not observed by the light, then there must be some occluding object in between, and we should draw that point as if it were in shadow.



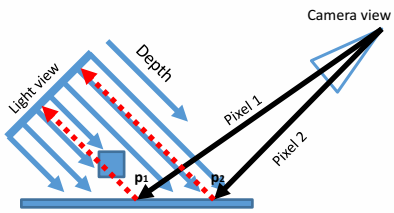


Shadow mapping

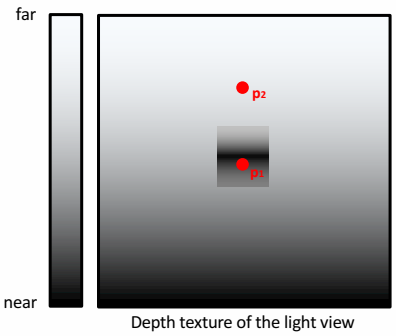
- It first generates a depth texture from the point of the light view.
- Then, we determine whether pixels are lit or not by using this depth texture.



Depth texture of the light view

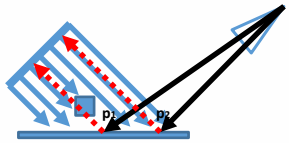


Occlusion decision



Say,
***d* : a depth texture value**
***z* : depth from light**

$d(p_1)<z(p_1) \Rightarrow p_1$ **occluded.**
$d(p_2)=z(p_2) \Rightarrow p_2$ **not occluded.**



Shadow mapping steps

- Rendering twice for one frame.

1. Render depth of a scene from the point of the light view.
   - The projection matrix of light could be orthogonal projection or perspective projection (directional light, point light).
2. Store a z-buffer of the light view as a depth texture (d).
3. Render a scene from the camera view point.
   - In vertex shader, we project 3D objects in the camera view and also project the 3D points of pixels into the light view to compute depth from light (z).
   - In the fragment shader, we compute pixels' depth from light and determine whether this point is lit or not by comparing pixels' depth (z) and correspond depth (d) values from the depth texture.

Shadow mapping

- In a first pass, we render into an FBO the scene as observed from some camera whose origin coincides with the position of the point light source. Let us model this camera transform as:

$$\begin{bmatrix} x_t w_t \\ y_t w_t \\ z_t w_t \\ w_t \end{bmatrix} = P_s M_s \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$

for appropriate matrices, P_s and M_s.

Shadow mapping

- During this first pass, we render the scene to an FBO using M_s as the modelview matrix and P_s as the projection matrix.
- In the FBO, we store, not the color of the point, but rather its z_t value.
- Due to z-buffering, the data stored at a pixel in the FBO represents the z_t value of the geometry closest to the light along the relevant line of sight. This FBO is then transferred to a texture.

Shadow mapping

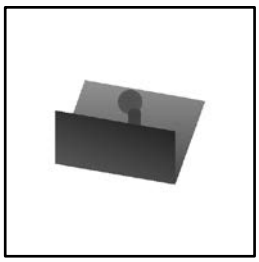
- During the second rendering pass, we render our desired image from the eye's point of view, but for each pixel, we check and see if the point we are observing was also observed by the light, or if it was blocked by something closer in the light's view.
- To do this, we use the same computation that was done with projector texture mapping
- Doing so, in the fragment shader, we can obtain the varying variables x_t, y_t and z_t associated with the point $[x_o, y_o, z_o, 1]^t$.

Shadow mapping

- We then compare this z_t value with the z_t value stored at $[x_t, y_t]^t$ in the texture.
- If these values agree then we are looking at a point that was also seen by the light; such a point is not in shadow and should be shaded accordingly. Conversely, if these values disagree, then the point we are looking at was occluded in the light's image, is in shadow and should be shaded as such.

Shadow acne

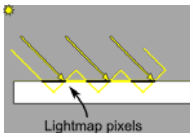
- Since a depth texture are discrete, acne artifact occurs in shadow.

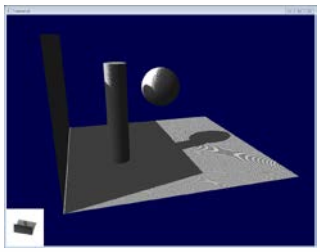


A depth texture

Pseudo code

```
If d(p)<z(p)
  visibility = 0.5
end
```





http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/

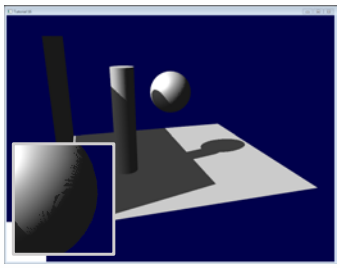
Solution for shadow acne

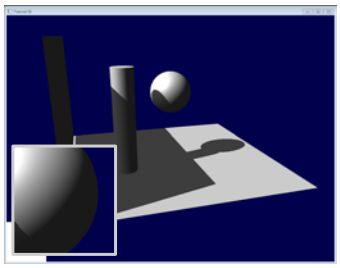
Constant bias

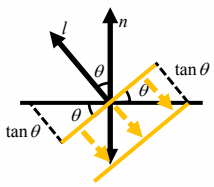
```
If d(p)+t<z(p)
  visibility= 0.5
end
```

Slope dependent bias

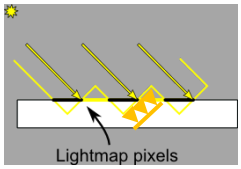
```
If d(p)+t*tanϑ <z(p)
  visibility = 0.5
end
```







Increase a shadow huddle.



http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/

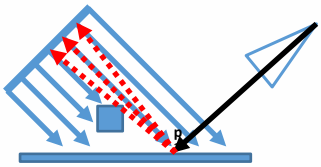
Shadow aliasing

- The depth texture are stored in a discrete manner, so it occurs aliasing artifact.
- To overcome this, we do multisampling.

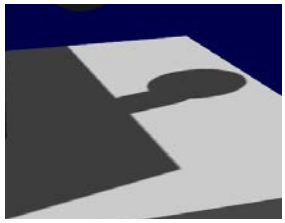
Pseudo code

```
For each sample i
  uv = p2uv(p) + offset_i
  If d(uv)+ t*tanϑ <z(p)
  shadow += 1
End
  visibility = 1 – 0.5*shadow/n
```

Aliasing artifact



Overview



A multisampling result