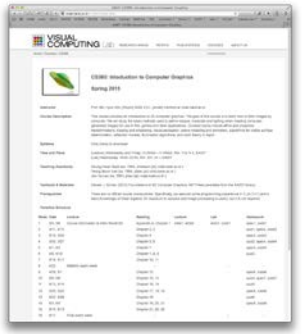
CS380: Introduction to Computer Graphics
Texture Mapping
Chapter 15

Min H. Kim
KAIST School of Computing

Announcements

- We will have quiz this week!



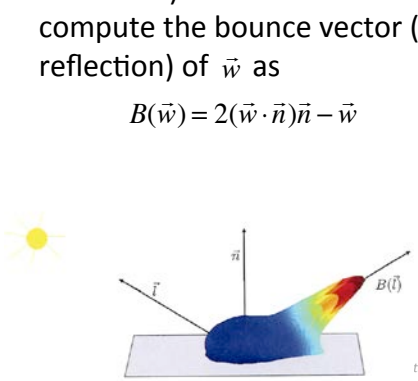
Materials

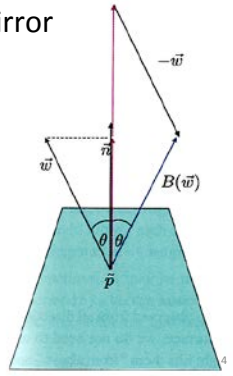
SUMMARY

Light blob from PVC plastic

- Recall: Given any vector $\vec{w}$ (not necessarily of unit norm) and a unit normal vector $\vec{n}$, we can compute the bounce vector (mirror reflection) of $\vec{w}$ as

$$B(\vec{w}) = 2(\vec{w} \cdot \vec{n})\vec{n} - \vec{w}$$

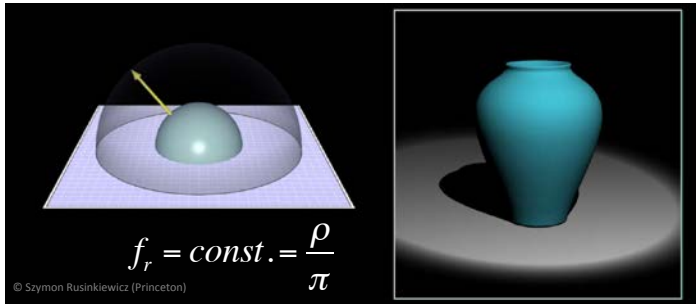




tler, 4

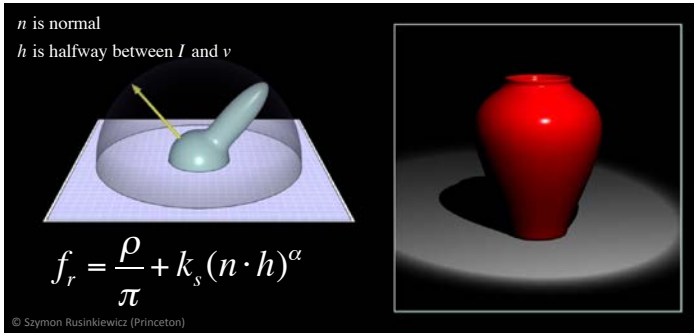
Diffuse reflectance model

- "Diffuse" materials, like rough wood, appear equally bright when observed from all direction $\vec{v}$
  - Thus, when calculating the color of a point on a diffuse surface, we do not need to use the $\vec{v}$ vector at all.



Blinn-Phong reflectance model

- Simple BRDF describing specular reflection, modeled in OpenGL

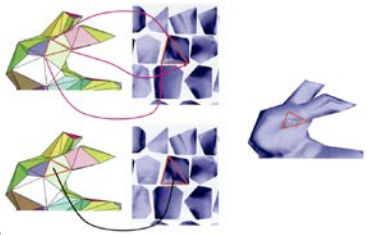


Chapter 15

TEXTURE MAPPING

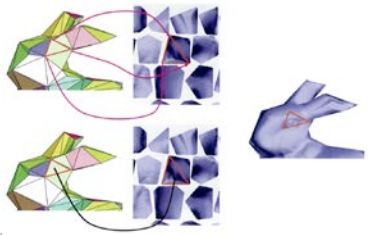
Texture mapping

- We have already seen and used texture mapping
- In basic texturing, we simply 'glue' part of an image onto a triangle by specifying texture coordinates at the three vertices.



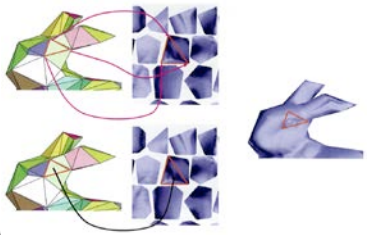
Texture mapping

- Bunch of OpenGL codes to load a texture and set various parameters (lin/const, mipmap, wrapping rules).
- A uniform variable is used to point to the desired texture unit.



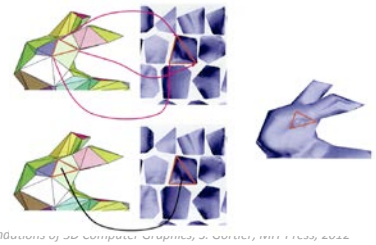
Texture mapping

- Varying variables are used to store texture coordinates.
- In this simplest incarnation, we just fetch r,g,b values from the texture and send them directly to the frame buffer.



Texture mapping

- Alternatively, the texture data could be interpreted as, say, the diffuse material color of the surface point, which would then be followed by the diffuse material computation described earlier.



Texture mapping

- initGLState()

```
…
glActiveTexture(GL_TEXTURE0);
glGenTextures(1, &h_texture);
glBindTexture(GL_TEXTURE_2D, h_texture);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
int twidth, theight;
packed_pixel_t * pixdata = ppmread("reachup.ppm", &twidth, &theight);
assert(pixdata);
glTexImage2D(GL_TEXTURE_2D, 0, GL_SRGB, twidth, theight, 0, GL_RGB,
GL_UNSIGNED_BYTE, pixdata);
free(pixdata);
…
```

Texture mapping

- initShaders()

```
h_texUnit0 = safe_glGetUniformLocation(h_program, "texUnit0");
h_aTexCoord = safe_glGetAttribLocation(h_program, "aTexCoord");
```

- display()

```
safe_glUniform1i(h_texUnit0, 0);
```

- Texture location (0,0) → lower left, (1,1) → upper right

```
GlfloatsqTex[12] =
{
  0, 0,
  1, 1,
  1, 0,

  0, 0,
  0, 1,
  1, 1
};
```


Texture mapping

- Vertex shader

```
#version 130
uniform float uVertexScale;
uniform mat4 uProjMatrix;
uniform mat4 uModelViewMatrix;
in vec2 aVertex;
in vec2 aTexCoord;
in vec3 aColor;
out vec3 vColor;
out vec2 vTexCoord;

void main()
{
  gl_Position = vec4(uProjMatrix * uModelViewMatrix * aVertex);
  vColor = aColor;
  vTexCoord = aTexCoord;
}
```


Texture mapping

- Fragment shader

```
#version 130

uniform sampler2D uTexUnit0;
in vec2 vTexCoord;
out vec4 fragColor;

void main() {
  vec4 texColor0 = texture2D(uTexUnit0, vTexCoord);
  fragColor = texColor0;
}
```


Normal mapping

- The data from a texture can also be interpreted in more interesting ways.
- In normal mapping, the r,g,b values from a texture are interpreted as the three coordinates of the normal at the point.
- This normal data can then be used as part of some material simulation,

Normal mapping

- Normal data has three coordinate values, each in the range [-1…1], while RGB textures store three values, each in the range [0…1] (0…255)
- So need some conversions:
  - R = normal_x/2.0 + 0.5;
  - normal_x = 2*R-1;
- Sometime need to deal with coordinate system conversions.

Environment cube maps

- Textures can also be used to model the environment in the distance around the object being rendered.
- In this case, we typically use 6 square textures representing the faces of a large cube surrounding the scene.

Environment cube maps

- Each texture pixel represents the color as seen along one direction in the environment.
- This is called a *cube map*. GLSL provides a cube-texture data type, samplerCube specifically for this purpose.

Environment cube maps

- During the shading of a point, we can treat the material at that point as a perfect mirror and fetch the environment data from the appropriate incoming direction.

Environment map shader

- We calculate $B(\vec{v})$ in the previous lecture.
- This bounced vector will point points towards the environment direction, which would be observed in a mirrored surface.
- By looking up the cube map, using this direction, we give the surface the appearance of a mirror.

Environment map shader

- Fragment shader

```
#version 130
uniform sampler2D uTexUnit0;
in vec3 nNormal;
in vec4 vPosition;
out vec4 fragColor;

vec3 reflect(vec3 w, vec3 n){
  return n*(dot(w,n)*2.0) - w; // bounce vector
}

void main() {
  vec3 normal = normalize(vNormal);
  vec3 reflected = reflect(normalize(vec3(-vPosition)), normal);
  vec4 texColor0 = textureCube(uTexUnit0, reflected);
  fragColor = vec4(texColor0.r, texColor0.g, texColor0.b, 1.0);;
}
```


Environment map shader

- -vPosition represents the view vector $\vec{v}$
- textureCube is a special GLSL function that takes a direction vector and returns the color stored at this direction in the cube texture map.
- Here we assume eye-coordinates, but frame changes may be needed.

Environment map shader

- This can be used for refraction.

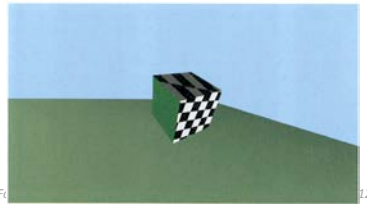
Projector texture mapping

- 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

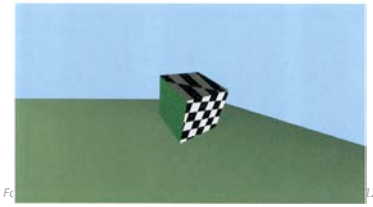
- The slide projector is modeled using 4 by 4, modelview and projection matrices, M_s and P_s

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



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;
}
```

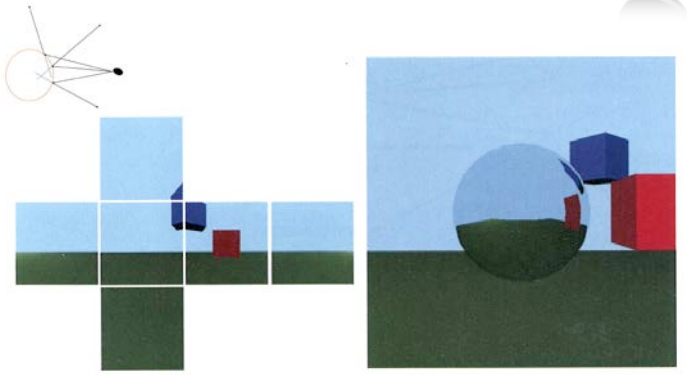

Projector texture mapping

- 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.

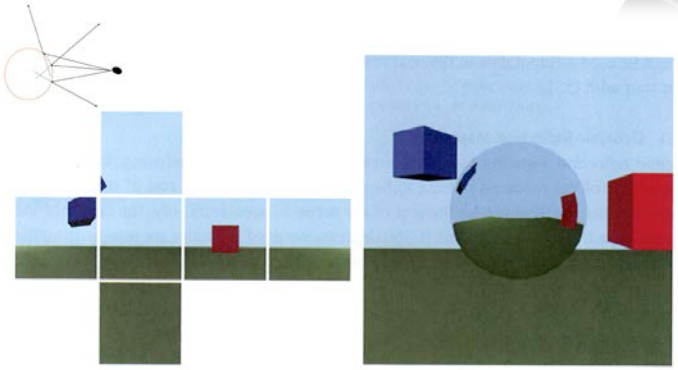
Multipass

- 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

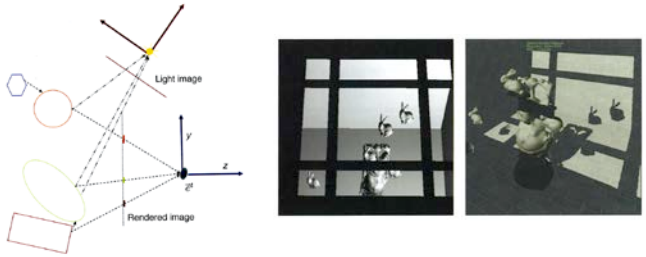


Multipass



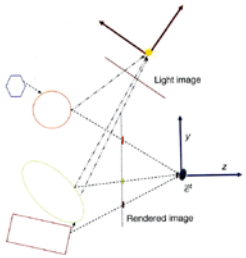
Shadow mapping

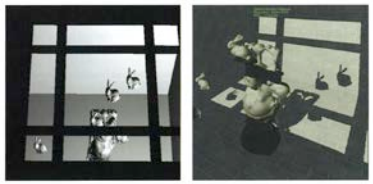
- The idea is to first create and store a z-buffered image from the point of view of the light, and then compare what we see in our view to what the light saw it its view.



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

- 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.