Recall: 6 Main Steps to Apply Texture

1. Create texture object
2. Specify the texture
   - Read or generate image
   - assign to texture (hardware) unit
   - enable texturing (turn on)
3. Assign texture (corners) to Object corners
4. Specify texture parameters
   - wrapping, filtering
5. Pass textures to shaders
6. Apply textures in shaders

still haven’t talked about setting texture parameters
Recall: Step 4: Specify Texture Parameters

- Texture parameters control how texture is applied
  - **Wrapping parameters** used if s, t outside (0,1) range
    - Clamping: if s, t > 1 use 1, if s, t < 0 use 0
    - Wrapping: use s, t modulo 1

```
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP )
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT )
```
**Magnification and Minification**

**Magnification:** Stretch small texture to fill many pixels

**Minification:** Shrink large texture to fit few pixels
Step 4: Specify Texture Parameters
Texture Value Lookup

How about coordinates that are not exactly at the intersection (pixel) positions?

A) Nearest neighbor
B) Linear Interpolation
C) Other filters
Example: Texture Magnification

- 48 x 48 image projected (stretched) onto 320 x 320 pixels

- Nearest neighbor filter
- Bilinear filter (avg 4 nearest texels)
- Cubic filter (weighted avg. 5 nearest texels)
Texture mapping parameters

1) Nearest Neighbor (lower image quality)

```c
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

2) Linear interpolate the neighbors
(better quality, slower)

```c
glTexParameteri(GL_TEXTURE_2D,
GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

Or

```c
GL_TEXTURE_MAX_FILTER
```
Dealing with Aliasing

- Point sampling of texture can lead to aliasing errors

miss blue stripes

point samples in texture space

point samples in $u,v$ (or $x,y,z$) space
Area Averaging

Better but slower option is area averaging
Other Stuff

- Wrapping texture onto curved surfaces. E.g. cylinder, can, etc
  \[ s = \frac{\theta - \theta_a}{\theta_b - \theta_a} \quad t = \frac{z - z_a}{z_b - z_a} \]

- Wrapping texture onto sphere
  \[ s = \frac{\theta - \theta_a}{\theta_b - \theta_a} \quad t = \frac{\phi - \phi_a}{\phi_b - \phi_a} \]

- Bump mapping: perturb surface normal by a quantity proportional to texture
Computer Graphics (CS 4731)
Lecture 20: Environment Mapping
(Reflections and Refractions)

Prof Emmanuel Agu
(Adapted from slides by Ed Angel)

Computer Science Dept.
Worcester Polytechnic Institute (WPI)
Environment Mapping

- Environmental mapping is a way to create the appearance of highly reflective and refractive surfaces without ray tracing.
Reflecting the Environment

Sphere of environment around object

Cube of environment around object
Types of Environment Maps

- Assumes environment infinitely far away
- Options: Store “object’s environment as

a) Sphere around object (sphere map)

b) Cube around object (cube map)

- OpenGL supports cube maps and sphere maps
Cube mapping

- Need to compute reflection vector, \( r \)
- Use \( r \) by for environment map lookup
Cube Map: How to Store

- Stores "environment" around objects as 6 sides of a cube (1 texture)
- Load 6 textures separately into 1 OpenGL cubemap
Cube Maps

- Loaded cube map texture can be accessed in GLSL through cubemap sampler

```
vec4 texColor = textureCube(mycube, texcoord);
```

- Texture coordinates must be 3D
Environment mapping

- projector function converts reflection vector \((x, y, z)\) to texture image \((u, v)\)

- viewer

- reflective surface

- environment texture image
Creating Cube Map

- Use 6 cameras directions from scene center
  - each with a 90 degree angle of view
Indexing into Cube Map

• Compute \( R = 2(N \cdot V)N - V \)
• Object at origin
• Perform lookup:

\[
\text{vec4 texColor} = \text{textureCube(mycube, R)};
\]

• Largest magnitude component of \( R \) \((x, y, z)\) used to determine face of cube
• Other 2 components give texture coordinates

More on this later....
Declaring Cube Maps in OpenGL

```c
glTextureMap2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X, level, rows, columns, border, GL_RGBA, GL_UNSIGNED_BYTE, image1)
```

- Repeat similar for other 5 images (sides)
- Make **1 cubemap texture object from 6 images**
- Parameters apply to all six images. E.g.

```c
glTexParameteri( GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAP_WRAP_S, GL_REPEAT)
```

- **Note:** texture coordinates are in 3D space (s, t, r)
Cube Map Example (init)

// colors for sides of cube
GLubyte red[3] = {255, 0, 0};
GLubyte green[3] = {0, 255, 0};
GLubyte blue[3] = {0, 0, 255};
GLubyte cyan[3] = {0, 255, 255};
GLubyte magenta[3] = {255, 0, 255};
GLubyte yellow[3] = {255, 255, 0};

GLenum enable(GL_TEXTURE_CUBE_MAP);

// Create texture object
glGenTextures(1, tex);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_CUBE_MAP, tex[0]);
Cube Map (init II)

Load 6 different pictures into 1 cube map of environment

gTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X ,
          0,3,1,1,0,GL_RGB,GL_UNSIGNED_BYTE, red);
gTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X ,
          0,3,1,1,0,GL_RGB,GL_UNSIGNED_BYTE, green);
gTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y ,
          0,3,1,1,0,GL_RGB,GL_UNSIGNED_BYTE, blue);
gTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y ,
          0,3,1,1,0,GL_RGB,GL_UNSIGNED_BYTE, cyan);
gTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z ,
          0,3,1,1,0,GL_RGB,GL_UNSIGNED_BYTE, magenta);
gTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z ,
          0,3,1,1,0,GL_RGB,GL_UNSIGNED_BYTE, yellow);
gTexImage2D(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER,GL_NEAREST);
GLEntex texMapLocation;
GLEntex tex[1];

texMapLocation = glGetUniformLocation(program, "texMap");
glUniform1i(texMapLocation, tex[0]);

Connect texture map (tex[0]) to variable texMap in fragment shader (texture mapping done in frag shader)
void quad(int a, int b, int c, int d) {
    static int i = 0;

    normal = normalize(cross(vertices[b] - vertices[a],
                              vertices[c] - vertices[b]));

    normals[i] = normal;
    points[i] = vertices[a];
    i++;

    // rest of data

    Calculate and set quad normals
Vertex Shader

out vec3 R;
in vec4 vPosition;
in vec4 Normal;
uniform mat4 ModelView;
uniform mat4 Projection;

void main() {
    gl_Position = Projection*ModelView*vPosition;
    vec4 eyePos  = vPosition;                   // calculate view vector V
    vec4 NN = ModelView*Normal;      // transform normal
    vec3 N =normalize(NN.xyz);            // normalize normal
    R = reflect(eyePos.xyz, N);              // calculate reflection vector R
}

Fragment Shader

in vec3 R;
uniform samplerCube texMap;

void main()
{
    vec4 texColor = textureCube(texMap, R); // look up texture map using R

    gl_FragColor = texColor;
}
Refraction using Cube Map

- Can also use cube map for refraction (transparent)
Reflection vs Refraction

Reflection

Refraction
Reflection and Refraction

- At each vertex

\[ I = I_{amb} + I_{diff} + I_{spec} + I_{refl} + I_{tran} \]

- Refracted component \( I_T \) is along transmitted direction \( t \)
Finding Transmitted (Refracted) Direction

- Transmitted direction obeys **Snell’s law**
- Snell’s law: relationship holds in diagram below

\[
\frac{\sin(\theta_2)}{c_2} = \frac{\sin(\theta_1)}{c_1}
\]

\(c_1, c_2\) are speeds of light in medium 1 and 2
Finding Transmitted Direction

- If ray goes from faster to slower medium (e.g. air to glass), ray is bent **towards** normal.
- If ray goes from slower to faster medium (e.g. glass to air), ray is bent **away** from normal.
- $c_1/c_2$ is important. Usually measured for medium-to-vacuum. E.g. water to vacuum.
- Some measured relative $c_1/c_2$ are:
  - Air: 99.97%
  - Glass: 52.2% to 59%
  - Water: 75.19%
  - Sapphire: 56.50%
  - Diamond: 41.33%
Transmission Angle

- Vector for transmission angle can be found as:

\[
\mathbf{t} = \frac{c_2}{c_1} \mathbf{dir} + \left( \frac{c_2}{c_1} (\mathbf{m} \cdot \mathbf{dir}) - \cos(\theta_2) \right) \mathbf{m}
\]

where

\[
\cos(\theta_2) = \sqrt{1 - \left( \frac{c_2}{c_1} \right) \left( 1 - (\mathbf{m} \cdot \mathbf{dir})^2 \right)}
\]

Or just use GLSL built-in function \textit{refract} to get \( T \)
out vec3 T;
in vec4 vPosition;
in vec4 Normal;
uniform mat4 ModelView;
uniform mat4 Projection;

void main() {
  gl_Position = Projection*ModelView*vPosition;
  vec4 eyePos = vPosition; // calculate view vector V
  vec4 NN = ModelView*Normal; // transform normal
  vec3 N = normalize(NN.xyz); // normalize normal
  T = refract(eyePos.xyz, N, iorefr); // calculate refracted vector T
}

Was previously
R = reflect(eyePos.xyz, N);
Refraction Fragment Shader

in vec3 T;
uniform samplerCube RefMap;

void main()
{
    vec4 refractColor = textureCube(RefMap, T);  // look up texture map using T
    refractcolor = mix(refractColor, WHITE, 0.3);  // mix pure color with 0.3 white

    gl_FragColor = refractcolor;
}
References

- Real Time Rendering by Akenine-Moller, Haines and Hoffman