EECS 487: Interactive Computer Graphics

Lecture 24:
• Texture Mapping

Texture Mapping

What is texture mapping
Texture mapping in OpenGL
• texture-coordinates array
Texture coordinates generation
Perspective-correct interpolation
Multitexture and Light Map
Texture mapping in GLSL

Surface Detail

What determines the “look” of a pixel?

Often results in 3D objects that look like “plastic objects floating in free space”

"If it looks like computer graphics, it is not good computer graphics”
– Jeremy Birn

“structured noise
pattern with randomness
section through volume
bumps

O’Brien
Texture Mapping

Instead, “glue on” a 2D image that captures the surface detail of the object

Modify the surface properties used in lighting computation without changing the underlying geometry, providing an illusion of detail

• combine fragment color with a lookup value
• or compute fragment color based on a lookup value

⇒ Image complexity doesn’t increase processing complexity

2D Texture Map

Texture is a 2D raster image: texture[width(s)][height(t)] of type RGB(A)

Texture coordinate (s, t) parameterized to [0, 1] range

Can be scaled to cover many different surfaces of arbitrary size and shape

Texture Mapping

Texture map: an array of values loaded from a file and stored in texture memory
• can be 1D, 2D, or 3D
• a unit of texture element is called a texel

Simplest case, texels contain scalar values:
• image texturing: surface color (RGB(A))

More generally, texels can also contain vectors:
• bump mapping: surface normals, to simulate apparent roughness
• environment mapping: reflection vectors, to simulate shiny and glossy surfaces

Procedural texture: instead of relying on a pre-computed lookup table, texturing can also be done algorithmically

2D Texture Mapping

Establish a mapping between surface point and texture

When shading a particular surface point
• look up corresponding texel in the texture image
• final color of point will be a function of the texel

Image complexity doesn’t increase processing complexity
Image-Based Rendering

Texture mapping in the extreme: using photos as textures to render dominant surfaces in scene

What You See Is ALL You Get (but that may be all you need)

Texture Coordinates

Assign texture coordinates to each vertex
- 1, 2, 3, or 4 texture dimensions per vertex
- index into the texture image, to retrieve texel corresponding to the vertex

Texture coordinates
- manually specified by programmer or automatically generated for every vertex
- interpolated during rasterization
- texturing itself done during fragment processing

Texture Mapping in OpenGL

1. Create a texture object:
   glGenTextures(), glBindTexture()
2. Specify a texture for that object: glTexImage2D()
   • optional:
     - gluScaleImage() // if dimensions are not powers of 2
     - glPixelStore*() // specify data format
3. Specify wrapping and filtering modes: glTexParameter*()
4. Specify how the texture is to be applied to each pixel:
   glTexEnv*()
5. Enable texture mapping: glEnable(GL_TEXTURE_2D)
6. Render the scene, supplying both geometric and texture coordinates: glTexCoord2f()

Creating a Texture Object

As with other OpenGL objects, first generate texture object descriptors*:

```c
int tods[N];
glGenTextures(N, tods)
// N is the number of texture objects to be allocated
// tods is an array to store the handles
```

Next, specify (by handle) which particular texture object to use for which type of texture and make it "current"

```c
glBindTexture(GL_TEXTURE_2D, tods[i]);
```

*texture descriptor == texture handle == texture name == texture ID
Specify the Texture Image

Specify the texture image to use:
```
glTexImage2D(target, level, internalFormat, width, height, border, format, type, teximage)
```
with:
- `target`: GL_TEXTURE_2D (or cube faces or others)
- `level`: mipmap level, 0 if not mipmapping
- `internalFormat`: GL_RGB or GL_RGBA, or a compressed format
- `width`: width of image, including border
- `height`: height of the image, including border
- `border`: whether image has border, must be 0 or 1
- `format`: format of image's pixel data, GL_RGB or GL_RGBA
- `type`: data type of pixel data, GL_UNSIGNED_BYTE, GL_FLOAT, etc.
- `teximage`: pointer to image or offset if pixel buffer object is used

Surface Larger than Texture

What if surface maps to \((s, t) > 1.0\) or \(< 0.0\) ?

Alternatives:
- repeat/wrap/tile
- mirror
- clamp to edge
- clamp (to border)

To repeat textures, use the fractional part of vertex coordinates as texture coordinates, for example: 5.3 → 0.3

In OpenGL use `glTexParameteri(*)` to specify alternative

Setting Texture Parameters

```
glTexParameteri(target, pname, param);
```
where
- `target` is GL_TEXTURE_2D
- `pname` is a parameter name that you want to change:
  - GL_TEXTURE_WRAP_T
  - GL_TEXTURE_WRAP_S
  - GL_TEXTURE_MIN_FILTER
  - GL_TEXTURE_MAG_FILTER
- `param` is the parameter value to change to

For example:
```
glTexParameterii(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

Texture Application Mode

```
glTexEnv(*);
```
tell OpenGL how each texture shall be combined with pre-existing fragment color

- GL_REPLACE: texture color replaces fragment color
  \[ c_f' = c_s, \alpha_f' = \alpha_s \]
- GL_ADD: \[ c_f' = c_f + c_s, \alpha_f' = \alpha_f + \alpha_s \]
- GL_MODULATE: multiply texture and fragment color
  \[ c_f' = c_f * c_s, \alpha_f' = \alpha_f * \alpha_s \]
- GL_BLEND: use texture value as blending value to blend fragment color and a predetermined color
  \[ c_f' = (1 - c_s) * c_f + c_s, \alpha_f' = \alpha_f * \alpha_s \]
- GL_DECAL: replace fragment color with texture color if texel is opaque
  \[ c_f' = (1 - \alpha_s) \cdot c_f + \alpha_s \cdot c_s, \alpha_f' = \alpha_f \]
Example: Diffuse Shading and Texture

**Want:** texture appear to be shaded, allowing for the perception of shape
- modulate texture only with diffuse light
- color the polygon white and light it normally
- use `glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE)`
- texture color is multiplied by surface (fragment) color, lowering texture brightness

**Problem:** modulating texture by light only makes it darker, we lost specular highlights!

**Solution:**
- separate out specular component as a secondary color

---

Rendering with Texture (in `init()`)

```c
/* First, read in the image file */
assert(fp = fopen("wood.ppm", "rb"));
for (i = 0 ; i < 256 ; i++)
    for (j = 0 ; j < 256 ; j++)
        for (k = 0 ; k < 3 ; k++) // RGB
            fscanf(fp, "%d %d %d", &teximage[i][j][k]);
fclose(fp);

/* Then set up the texture */
int tod;
int texCoord[4];
glGenTextures(1, &tod);
glBindTexture(GL_TEXTURE_2D, tod);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

Rendering with Different Textures

```c
// in init():
glBindTexture(GL_TEXTURE_2D, textures[0]);
gl一族Parameteri(...); ... ; glTexImage2D(GL_TEXTURE_2D,...);

// in display():
glBindTexture(GL_TEXTURE_2D, textures[0]);
gBegin(...);
gTexCoord (...);
gVertex  (...);
gEnd {...};
```
Texture-Coordinates Array

When a vertex array is used, texture coordinates corresponding to the vertices must be provided in a texture-coordinates array (see Lab 6)

```c
// texcoords must have a 1-1 mapping with vertices
float vertices[][] = { { 0.5, 0.5, 0.0 },
    {0.5, -0.5, 0.0 }};
float texcoords[][] = { { 1.0, 1.0 },
    { 0.0, 0.0 }, { 1.0, 0.0 } };;
```

```c
glEnableClientState(GL_VERTEX_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glTexCoordPointer(2, GL_FLOAT, 0, texcoords);
```

Texture Coordinates Autogen

How do we “paste” a 2D texture image onto a 3D object?

Non-parametrically
- texture size and orientation are fixed in world coordinates
- gives a “projector” effect: object “swims” through texture

Parametrically
- texture size and orientation tied to object, in object coordinates
- map object coordinates to texture coordinates

Planar Mapping

How do we map to polygonal meshes?

Planar/orthographic map:
- simply remove one of the object’s coordinates to project onto that coordinate plane
- the texture is constant in one direction (z, x, y)

Cylindrical Map

Object coordinate \((x, y, z)\) is converted to \((r(\text{adius}), \theta, h(\text{eight}))\)

For texture mapping, \(\theta\) is converted into \(s\)-coordinate and \(h\) is converted into \(t\)-coordinate

This wraps the texture map around the object

Useful for faces
Cylindrical Map

At minimum and maximum extents of the cylinder, the texture gets pinched together.

Spherical Map

Convert from \((x, y, z)\) to spherical coordinates \((\theta, \varphi)\).

Longitude \((\varphi)\) is converted into \(s\)-coordinate, latitude \((\theta)\) is converted into \(t\)-coordinate (note \(z\) is not pointing up in image).

Spherical Map

Not only pinches the texture at the poles, but also stretches the squares along the equator.

Cube/Box Map

Use six planar maps, one for each face of the cube.
Generating Texture Coordinates

OpenGL can generate texture coordinates automatically using `glTexGen*()`:
- based on distance of vertex from a given plane in either
  - object-coordinates (GL_OBJECT_LINEAR): texture attached to object, or
  - eye-coordinates (GL_EYE_LINEAR): object appears swimming in texture, e.g., to render an oil drill, as it goes deeper into ground, it changes color.

Intermediate to Object Mapping

<table>
<thead>
<tr>
<th></th>
<th>Plane</th>
<th>Cylinder</th>
<th>Sphere</th>
<th>Box</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Normal</td>
<td>-</td>
<td>x</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>Object Centroid</td>
<td>-</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Intermediate Surface Normal</td>
<td>slide projector</td>
<td>shrinkwrap</td>
<td>-</td>
<td>ok</td>
</tr>
<tr>
<td>Reflected ray</td>
<td>EM</td>
<td>EM</td>
<td>EM</td>
<td>EM</td>
</tr>
</tbody>
</table>

Texture Mapping with an Intermediate Surface

Two-stage mapping:
1. map the texture to a simple intermediate surface (cube, cylinder, sphere)
2. map the intermediate surface (with the texture) onto the surface being rendered

Intermediate to Object Mapping

<table>
<thead>
<tr>
<th></th>
<th>Plane</th>
<th>Cylinder</th>
<th>Sphere</th>
<th>Box</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Normal</td>
<td>-</td>
<td>x</td>
<td>ok</td>
<td>ok</td>
</tr>
<tr>
<td>Object Centroid</td>
<td>-</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Intermediate Surface Normal</td>
<td>slide projector</td>
<td>shrinkwrap</td>
<td>-</td>
<td>ok</td>
</tr>
<tr>
<td>Reflected ray</td>
<td>EM</td>
<td>EM</td>
<td>EM</td>
<td>EM</td>
</tr>
</tbody>
</table>

Rasterizing Texture Coordinates

When rasterizing primitives:
- assign texture coordinates to each vertex
- within a triangle, use linear interpolation (barycentric coordinates!)
Perspective Projection

Characteristics preserved:
- Rigid body/Euclidean: angles, lengths, areas
- Similitudes/similarity: angles, length ratios
- Affine: parallel lines, length ratios, area ratios
- Perspective: collinearity, cross-ratios

Linear Interpolation in Perspective

Linear interpolation in screen coordinates is not equal to linear interpolation in eye coordinates!

\[
p_o = p_0 + \alpha(p_1 - p_0), \ s_o \neq 0.5
\]

Solution?

Perspective-correct Interpolation

Instead of interpolating the parameter \( s' \) after perspective divide, do the perspective divide on the interpolated \( s \) and interpolated homogenous coordinate \( w \):
- \( s_0 = \text{lerp}(s_0, s_1) \)
- \( w_0 = \text{lerp}(w_0, w_1) \)
- \( s'_o = s_o/w_o \)

Bilinear Interpolation

Linear interpolation in 2D:
\[
e = (1-u)a + ub
\]
\[
f = (1-u)c + ud
\]
\[
p(u,v) = (1-v)e + vf = (1-u)(1-v)a + u(1-v)b + (1-u)v c + uv d
\]

Also requires perspective correction:

Heckbert89

Heckbert89
Bilinear Interpolation in Perspective

Uncorrected, not only lack of foreshortening, worse effect if square is rotated:

Effect is most visible on texture mapping, but also presents in color shading, though generally tolerated

Perspective-correct interpolation in OpenGL:

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
```

Multitexture Pipeline

Applying multiple textures to a single fragment
- applied one by one in a pipelined fashion
- each stage consists of a texture unit/environment
- allows for texture blending, for lighting effects, decals, compositing

```
| C0 | TE0 | TE1 | TE2 | C1 |
```

\[ C_f = \text{fragment primary color input to texturing} \]
\[ C_i = \text{fragment color output from texturing} \]
\[ C_t = \text{texture color from texture lookup} \]
\[ T_E = \text{texture environment} \]

Text Units

A texture unit allows texture binding to be encapsulated into a single texture context/environment

In OpenGL, there are only a fixed, pre-determined number (≥80) of texture units and they are not dynamic OpenGL objects

To select a particular texture unit, use

```
glActiveTexture();
```

• after which calls to `glBindTexture()`, affect only the selected texture unit

Using Multitexture Pipeline

```
// In init(), load images and initialize textures as before.
// In display():

// bind and enable texture unit 0

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, tbods[0]);
glEnable(GL_TEXTURE_2D);

// bind and enable texture unit 1

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, tbods[1]);
glEnable(GL_TEXTURE_2D);

// specify two sets of texture coordinates per vertex

glBegin(GL_TRIANGLES);

glColor3f(1.0f, 1.0f, 1.0f);

```

\[ \text{glMultiTexCoord2f(GL_TEXTURE0, 0.0, 1.0);} \]
\[ \text{glMultiTexCoord2f(GL_TEXTURE1, 0.0, 1.0);} \]
\[ \text{glVertex3f(...);} \]

```
glEnd();
```
Texture sampler

A texture unit is passed, as a uniform variable, from application to shaders as a texture sampler.

Fragment shaders use texture*() to sample texture from sampler*

Example: to pass texture unit 0 from application to shader as a sampler, assuming texture unit 0 has been set up as shown previously:

```c
GLuint tid;
tid = glGetUniformLocation(myprog, "mytexture");
glUniform1i(tid, 0) // assign texture object and texture unit 0 to sampler
```

Interpolated Texture Coordinates

// vertex shader: generally only worries about texcoords

```c
uniform vec4 lightPos;
varying vec3 normal;
varying vec3 lightVec;
varying vec2 texcoord;

void main() {
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    texcoord = 0.01*gl_Vertex.xz;
    vec4 vert = gl_ModelViewMatrix * gl_Vertex;
    normal = gl_NormalMatrix * gl_Normal;
    lightVec = vec3(lightPos - vert);
}
```

Texture Coordinates as GLSL 1.3+ Custom Vertex Attribute

```c
attribute vec4 va_Position;
attribute vec2 va_TexCoords;
varying texcoords;

void main(void) {
    gl_Position = gl_ModelViewProjectionMatrix*va_Position;
    texcoords = va_TexCoords;
}
```

Application loads, compiles, and links shaders as usual

Then application gets the cva locations:

```c
int vPos = glGetAttribLocation(pd, "va_Position");
int vTex = glGetAttribLocation(pd, "va_TexCoords");
```
Binding CVA with Data Stream

Setup vbo GL_ARRAY_BUFFER to include texture coordinates (see PA3)

Enable cva:

   glEnableVertexAttribArray(vTex);

Then bind the cva to the vbo holding the data stream:

   glVertexAttribPointer(vTex, 2, GL_FLOAT, GL_FALSE,
                        stride, offset);