Texture Mapping

CS 432 Interactive Computer Graphics
Prof. David E. Breen
Department of Computer Science

Objectives

- Introduce Mapping Methods
  - Texture Mapping
  - Environment Mapping
  - Bump Mapping
- Consider basic strategies
  - Forward vs backward mapping
  - Point sampling vs area averaging

The Limits of Geometric Modeling

- Although graphics cards can render over 20 million polygons per second, that number is insufficient for many phenomena
  - Clouds
  - Grass
  - Terrain
  - Skin

Modeling an Orange

- Consider the problem of modeling an orange (the fruit)
- Start with an orange-colored sphere
  - Too simple
- Replace sphere with a more complex shape
  - Does not capture surface characteristics (small dimples)
  - Takes too many polygons to model all the dimples

Modeling an Orange (2)

- Take a picture of a real orange, scan it, and “paste” onto simple geometric model
  - This process is known as texture mapping
- Still might not be sufficient because resulting surface will be smooth
  - Need to change local shape
  - Bump mapping

Three Types of Mapping

- Texture Mapping
  - Uses images to fill inside of polygons
- Environment (reflection mapping)
  - Uses a picture of the environment for texture maps
  - Allows simulation of highly specular surfaces
- Bump mapping
  - Emulates altering normal vectors during the rendering process
Texture Mapping

- geometric model
- texture mapped

Environment Mapping

- Mapping techniques are implemented at the end of the rendering pipeline
- Very efficient because few polygons make it past the clipper

Bump Mapping

- Where does mapping take place?
- Parametric coordinates
  - May be used to model curves and surfaces
- Texture coordinates
  - Used to identify points in the image to be mapped
- Object or World Coordinates
  - Conceptually, where the mapping takes place
- Window Coordinates
  - Where the final image is really produced

Is it simple?

- Although the idea is simple—map an image to a surface—there are 3 or 4 coordinate systems involved

Coordinate Systems
Texture Mapping

parametric coordinates

Texture coordinates

World coordinates

Window coordinates

Mapping Functions

- Basic problem is how to find the maps
- Consider mapping from texture coordinates to a point on the surface
- Appear to need three functions:
  \[ x = x(s,t) \]
  \[ y = y(s,t) \]
  \[ z = z(s,t) \]
- But we really want to go the other way:
  \[ s = s(x,y,z) \]
  \[ t = t(x,y,z) \]

Backward Mapping

- We really want to go backwards:
  - Given a pixel, we want to know to which point on an object it corresponds.
  - Given a point on an object, we want to know to which point in the texture it corresponds.
- Need a map of the form:
  \[ s = s(x,y,z) \]
  \[ t = t(x,y,z) \]
- Such functions are difficult to find in general.

Two-part mapping

- One solution to the mapping problem is to first map the texture to a simple intermediate surface.
- Example: map to cylinder

Cylindrical Mapping

parametric cylinder

\[ x = r \cos 2\pi u \]
\[ y = r \sin 2\pi u \]
\[ z = v/h \]

maps rectangle in \( uv \) space to cylinder of radius \( r \) and height \( h \) in world coordinates

\[ s = u \]
\[ t = v \]

maps from texture space

Spherical Map

We can use a parametric sphere

\[ x = r \cos 2\pi u \]
\[ y = r \sin 2\pi u \cos 2\pi v \]
\[ z = r \sin 2\pi u \sin 2\pi v \]

in a similar manner to the cylinder but have to decide where to put the distortion.

Spheres are used in environmental maps.
Box Mapping

- Easy to use with simple orthographic projection
- Also used in environment maps

Second Mapping

- Map from intermediate object to actual object
  - Normals from intermediate to actual
  - Normals from actual to intermediate
  - Vectors from center of intermediate

Aliasing

- Point sampling of the texture can lead to aliasing errors
- Point samples in u,v (or x,y,z) space
- Point samples in texture space
- Miss blue stripes

Area Averaging

- A better but slower option is to use area averaging
- Note that preimage of pixel is curved

WebGL Texture Mapping

Go to ART Texture Mapping slides
Objectives

• Introduce WebGL texture mapping
  - two-dimensional texture maps
  - assigning texture coordinates
  - forming texture images

Basic Strategy

Three steps to applying a texture
1. specify the texture
   • read or generate image
   • assign to texture
   • enable texturing
2. assign texture coordinates to vertices
   • Proper mapping function is left to application
3. specify texture parameters
   • wrapping, filtering

Texture Mapping

Texture Example

• The texture (below) is a 256 x 256 image that has been mapped to a rectangular polygon which is viewed in perspective

Texture Mapping and the WebGL Pipeline

• Images and geometry flow through separate pipelines that join during fragment processing
  - “complex” textures do not affect geometric complexity

Specifying a Texture Image

• Define a texture image from an array of texels (texture elements) in CPU memory
• Use an image in a standard format such as JPEG
  - Scanned image
  - Generate by application code
• WebGL supports only 2 dimensional texture maps
  - no need to enable as in desktop OpenGL
  - desktop OpenGL supports 1-4 dimensional texture maps
### Define Image as a Texture

```javascript
gl.texImage2D( target, level, internalformat, w, h, border, format, type, texels );
```

- **target**: type of texture, e.g. `gl.TEXTURE_2D`
- **level**: used for mipmapping (discussed later)
- **internalformat**: elements per texel
- **w, h**: width and height of texels in pixels
- **border**: width of the border. Must be 0.
- **format**: must be the same as `internalformat`
- **type**: data type of texel data
- **texels**: Source of texel data

```javascript
glTexImage2D(gl.TEXTURE_2D, 0, gl.RGB, 512, 512, 0, gl.RGB, gl.UNSIGNED_BYTE, my_texels);
```

### A Checkerboard Image

```javascript
var image1 = new Uint8Array (4*texSize*texSize);
for ( var i = 0; i < texSize; i++ ) {
  for ( var j = 0; j < texSize; j++ ) {
    var patchx = Math.floor(i/(texSize/numChecks));
    var patchy = Math.floor(j/(texSize/numChecks));
    if (patchx%2 ^ patchy%2) c = 255 ;
    else c = 0;
    //c = 255*((i & 0x8) == 0) ^ ((j & 0x8)  == 0))
    image1[4*i*texSize+4 *j] = c;
    image1[4*i*texSize+4 *j+1] = c;
    image1[4*i*texSize+4 *j+2] = c;
    image1[4*i*texSize+4 *j+3] = 255;
  }
}
```

### Using a GIF image

```javascript
// specify image in JS file
var image = new Image();
image.src = "SA2011_black.gif"
window.onload = function() {
  configureTexture( image );
}
```

### Cube Example

```javascript
var texCoord = [
  vec2(0, 0),
  vec2(0, 1),
  vec2(1, 1),
  vec2(1, 0)
];
```

- **Based on parametric texture coordinates**
- **Specify as a 2D vertex attribute**

### Mapping a Texture

- Based on parametric texture coordinates
- Specify as a 2D vertex attribute

### Interpolation

- WebGL uses interpolation to find proper texels from specified texture coordinates
- Can be distortions
  - good selection of tex coordinates
  - poor selection of tex coordinates
  - texture stretched over trapezoid showing effects of bilinear interpolation
Using Texture Objects

1. Specify textures in texture objects
2. Set texture filter
3. Set texture function
4. Set texture wrap mode
5. Set optional perspective correction hint
6. Bind texture object
7. Enable texturing
8. Supply texture coordinates for vertex
   - Coordinates can also be generated

Texture Parameters

- WebGL has a variety of parameters that determine how texture is applied
  - Wrapping parameters determine what happens if s and t are outside the (0,1) range
  - Filter modes allow us to use area averaging instead of point samples
  - Mipmapping allows us to use textures at multiple resolutions
  - Environment parameters determine how texture mapping interacts with shading

Wrapping Mode

Clamping: if \( s, t > 1 \) use 1, if \( s, t < 0 \) use 0
Wrapping: use \( s, t \) modulo 1

\[
\begin{align*}
gl & . texParameteri ( gl . TEXTURE_2D, \\
gl & . TEXTURE_WRAP_S, gl . CLAMP ) \\
gl & . texParameteri ( gl . TEXTURE_2D, \\
gl & . TEXTURE_WRAP_T, gl . REPEAT ) \\
\end{align*}
\]

Filter Modes

Modes determined by

\[
\begin{align*}
gl & . texParameteri ( \text{target}, \text{type}, \text{mode} ) \\
gl & . texParameteri ( gl . TEXTURE_2D, gl . TEXTURE_MAG_FILTER, \\
gl & . NEAREST ) ; \\
gl & . texParameteri ( gl . TEXTURE_2D, gl . TEXTURE_MIN_FILTER, \\
gl & . LINEAR ) ; \\
\end{align*}
\]

Magnification and Minification

More than one texel can cover a pixel (minification) or more than one pixel can cover a texel (magnification)

Can use point sampling (nearest texel) or linear filtering (2 x 2 filter) to obtain texture values

Mipmapped Textures

- Allows for prefiltered texture maps of decreasing resolutions
- Lessens interpolation errors for smaller textured objects
- Declare mipmap level during texture definition

\[
\begin{align*}
gl & . texImage2D ( gl . TEXTURE_*D, \text{level} , ) \\
\end{align*}
\]

- Have WebGL make your mipmap with

\[
\begin{align*}
gl & . generateMipmap () \\
\end{align*}
\]
Example

point sampling

linear filtering

mipmapped point sampling

mipmapped linear filtering

Other Texture Features

- **Environment Maps**
  - Start with image of environment through a wide angle lens
  - Can be either a real scanned image or an image created in WebGL
  - Use this texture to generate a spherical map
  - Alternative is to use a cube map

- **Multitexturing**
  - Apply a sequence of textures through cascaded texture units

Using Textures

- Texture value may be used in ANY of the components of the shading formula
- For example
  - Diffuse color
  - Specular color
  - Ambient color
  - Shininess
  - Normals
  - Alpha
- Or as a decal, or mask or blended in

Typical Code

- Application sending texture coordinates
  
  ```
  var vPosition = gl.getAttribLocation( program, "vPosition" );
  gl.enableVertexAttribArray( vPosition );
  gl.vertexAttribPointer( vPosition, 4, gl.FLOAT, false, 0, 0 );
  
  gl.bindBuffer( gl.ARRAY_BUFFER, tBuffer );
  var vTexCoord = glGetAttribLocation( program, "vTexCoord" );
  gl.enableVertexAttribArray( vTexCoord );
  gl.vertexAttribPointer( vTexCoord, 2, gl.FLOAT, false, 0, 0 );
  ```

Vertex Shader

- Usually vertex shader will output texture coordinates to be rasterized
- Must do all other standard tasks too
  - Compute vertex position
  - Compute vertex color if needed

```
attribute vec4 vPosition; // vertex position in object coordinates
attribute vec4 vColor; // vertex color from application
attribute vec2 vTexCoord; // texture coordinate from application

varying vec4 color; // output color to be interpolated
varying vec2 texCoord; // output tex coordinate to be interpolated
```
Applying Textures

• Textures are applied during fragments shading by a sampler
• Samplers return a texture color from a texture object

```c
void main()
{
  gl_FragColor = color * texture2D(texture, texCoord);
}
```

Texture Object

```c
function configureTexture( image ) {
  var texture = gl.createTexture();
  gl.bindTexture( gl.TEXTURE_2D, texture );
  gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
  gl.texImage2D( gl.TEXTURE_2D, 0, gl.RGB, gl.RGB,
                gl.UNSIGNED_BYTE, image );
  gl.generateMipmap( gl.TEXTURE_2D );
  gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
                   gl.NEAREST_MIPMAP_LINEAR );
  gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
                   gl.NEAREST );
  gl.activeTexture(gl.TEXTURE0);
  gl.uniform1i(gl.getUniformLocation(program, "texture"), 0);
}
```

Linking with Shaders

```c
var vTexCoord = gl.getAttribLocation( program, "vTexCoord" );
gl.enableVertexAttribArray( vTexCoord );
// Set the value of the fragment shader texture sampler variable
// ("texture") to the the appropriate texture unit. In this case,
// zero for GL_TEXTURE0 which was previously set by calling
// gl.activeTexture().

// Set the value of the fragment shader texture sampler variable
// ("texture") to the the appropriate texture unit. In this case,
// zero for GL_TEXTURE0 which was previously set by calling
// gl.activeTexture().

gl.uniform1i(glGetUniformLocation(program, "texture"), 0);
```

Reflection and Environment Maps

```c
var vTexCoord = gl.getAttribLocation( program, "vTexCoord" );
```

Objectives

• Texture Mapping Applications
• Reflection (Environment) Maps
  - Cube Maps
  - Spherical Maps
• Bump Maps

Mapping Variations

smooth shading  environment mapping  bump mapping
Environment Mapping

- Environmental (reflection) mapping is a way to create the appearance of highly reflective surfaces without ray tracing which requires global calculations.
- Introduced in movies such as The Abyss and Terminator 2.
- Prevalent in video games.
- It is a form of texture mapping.

Reflecting the Environment

Mapping to a Sphere

- If we map all objects to hemisphere, we cannot tell if they are on the sphere or anywhere else along the reflector.
- Use the map on the sphere as a texture that can be mapped onto the object.
- Can use other surfaces as the intermediate.
  - Cube maps
  - Cylinder maps

Hemisphere Map as a Texture

Cube Map

- Compute \( R = 2(N \cdot V)N - V \).
- Object at origin.
- Use largest magnitude component \( V \) of \( R \) to determine face of cube.
- Other two components give texture coordinates.

Indexing into Cube Map
WebGL Implementation

- WebGL supports only cube maps
  - desktop OpenGL also supports sphere maps
- First must form map
  - Use images from a real camera
  - Form images with WebGL
- Texture map it to object

Cube Maps

- We can form a cube map texture by defining six 2D texture maps that correspond to the sides of a box
- Supported by WebGL through cubemap sampler
  ```glsl
textColor = textureCube(mycube, texcoord);
```
- Texture coordinates must be 3D
  - usually are given by the vertex location so we don't need compute separate tex coords

Cube Maps

- We can form a cube map texture by defining six 2D texture maps that correspond to the sides of a box
- Supported by WebGL through cubemap sampler
- Texture coordinates must be 3D
- Usually are given by the vertex location so we don't need compute separate tex coords

Environment Maps with Shaders

- Environment maps are usually computed in world coordinates which can differ from object coordinates because of the modeling matrix
  - May have to keep track of modeling matrix and pass it to the shaders as a uniform variable
- Can also use reflection map or refraction map for effects such as simulating water

Issues

- Must assume environment is very far from object (equivalent to the difference between near and distant lights)
- Object cannot be concave (no self reflections possible)
- No reflections between objects
- Need a reflection map for each object
- Need a new map if viewer moves

Forming Cube Map

- Use six cameras, each with a 90 degree angle of view
Doing it in WebGL

```javascript
gl.textureMap2D(
    gl.TEXTURE_CUBE_MAP_POSITIVE_X,
    level, rows, columns, border, gl.LRGB, gl.UNSIGNED_BYTE, image1)

• Same for other five images
• Make one texture object out of the six images
```

Example

```javascript
• Consider rotating cube inside a cube that reflects the color of the walls
• Each wall is a solid color (red, green, blue, cyan, magenta, yellow)

  var red = new Uint8Array([255, 0, 0, 255]);
  var green = new Uint8Array([0, 255, 0, 255]);
  var blue = new Uint8Array([0, 0, 255, 255]);
  var cyan = new Uint8Array([0, 255, 255, 255]);
  var magenta = new Uint8Array([255, 0, 255, 255]);
  var yellow = new Uint8Array([255, 255, 0, 255]);
```

Texture Object

```javascript
cubeMap = gl.createTexture();
bindTexture(gl.TEXTURE_CUBE_MAP, cubeMap);
gl.textureImage2D(gl.TEXTURE_CUBE_MAP_POSITIVE_X, 0, gl.RGBA,
1, 1, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, red);
gl.textureImage2D(gl.TEXTURE_CUBE_MAP_NEGATIVE_X, 0, gl.RGBA,
1, 1, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, green);
gl.textureImage2D(gl.TEXTURE_CUBE_MAP_POSITIVE_Y, 0, gl.RGBA,
1, 1, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, blue);
gl.textureImage2D(gl.TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, gl.RGBA,
1, 1, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, cyan);
gl.textureImage2D(gl.TEXTURE_CUBE_MAP_POSITIVE_Z, 0, gl.RGBA,
1, 1, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, yellow);
gl.textureImage2D(gl.TEXTURE_CUBE_MAP_NEGATIVE_Z, 0, gl.RGBA,
1, 1, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, magenta);
gl.activeTexture(gl.TEXTURE0);
gl.uniform1i(gl.getUniformLocation(program, "texMap"), 0);
```

Vertex Shader

```javascript
varying vec3 R;
attribute vec4 vPosition;
attribute vec4 vNormal;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform vec3 theta;
void main(){
    vec3 angles = radians(theta);
   // compute rotation matrices rx, ry, rz here
   mat4 ModelViewMatrix = ModelViewMatrix * rz * ry * rx;
   gl_Position = projectionMatrix * ModelViewMatrix * vPosition;
   vec4 eyePos = ModelViewMatrix * vPosition;
   vec3 N = ModelViewMatrix * vNormal;
   R = reflect(eyePos.xyz, N.xyz);
}
```

Fragment Shader

```javascript
precision mediump float;

varying vec3 R;
uniform samplerCube texMap;
void main(){
    vec4 textColor = textureCube(texMap, R);
    gl_FragColor = textColor;
}
```

Sphere Mapping

• Original environmental mapping technique proposed by Blinn and Newell based in using lines of longitude and latitude to map parametric variables to texture coordinates
• OpenGL supports sphere mapping which requires a circular texture map equivalent to an image taken with a fisheye lens
13

**Sphere Map**

**Bump Maps**

Ed Angel  
Professor Emeritus of Computer Science  
University of New Mexico

**Objectives**

- Introduce bump mapping

**Modeling an Orange**

- Consider modeling an orange  
- Texture map a photo of an orange onto a surface  
  - Captures dimples  
  - Will not be correct if we move viewer or light  
  - We have shades of dimples rather than their correct orientation  
- Ideally we need to perturb normal across surface of object and compute a new color at each interior point

**Bump Mapping (Blinn)**

- Consider a smooth surface

**Rougher Version**

- Consider a smooth surface  
  - Normal at point $p$  
  - Normal perturbed to $n'$
### Tangent Plane

![Tangent Plane Diagram]

- **Equations**
  
  \[ p(u,v) = [x(u,v), y(u,v), z(u,v)]^T \]

  \[ p_u = [\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}]^T \]

  \[ p_v = [\frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v}]^T \]

  \[ n = (p_u \times p_v) / |p_u \times p_v| \]

### Displacement Function

- **Equations**

  \[ p' = p + d(u,v) \cdot n \]

  \[ d(u,v) \text{ is the bump or displacement function} \]

  \[ |d(u,v)| << 1 \]

### Perturbed Normal

- **Equations**

  \[ n' = p'_u \times p'_v \]

  \[ p'_u = p_u + (\frac{\partial d}{\partial u}) \cdot n + d(u,v) \cdot n_u \]

  \[ p'_v = p_v + (\frac{\partial d}{\partial v}) \cdot n + d(u,v) \cdot n_v \]

  If \( d \) is small, we can neglect last term

### Approximating the Normal

- **Equations**

  \[ n' = p'_u \times p'_v \]

  \[ n' \approx n + (\frac{\partial d}{\partial u}) \cdot n \times p_v + (\frac{\partial d}{\partial v}) \cdot n \times p_u \]

  The vectors \( n \times p_v \) and \( n \times p_u \) lie in the tangent plane

  Hence the normal is displaced in the tangent plane

  Must precompute the arrays \( \frac{\partial d}{\partial u} \) and \( \frac{\partial d}{\partial v} \)

  Finally, we perturb the normal during shading

### Image Processing

- **Suppose that we start with a function**
  \( d(u,v) \)

- **We can sample it to form an array**
  \( D = [d_{ij}] \)

- **Then**
  \[ \frac{\partial d}{\partial u} \equiv d_{ij} - d_{i-1,j} \]

  and

  \[ \frac{\partial d}{\partial v} \equiv d_{ij} - d_{i,j-1} \]

- **Embossing:** multipass approach using floating point buffer
How to do this?

- The problem is that we want to apply the perturbation at all points on the surface
- Cannot solve by vertex lighting (unless polygons are very small)
- Really want to apply to every fragment
- Can't do that in fixed function pipeline
- But can do with a fragment program!!

See bumpmap.html and bumpmap.js

Suggestions for HW8

- First add texture coordinates \([u,v]\) to your vertices
- Just use the \([u,v]\) pair used to compute the vertex on the Bezier patch
- Use a variant of the code in slides 45 -> 49 to define your texture object in your application and fragment shader
- Note that the texture coordinates get passed through to the fragment shader, where they are used to sample the 2D texture

Suggestions for HW8

- To test if your texture coordinates are getting to the shaders properly, you could simply map the coordinates to the red & green channels of the fragment's color
- This is one way to visualize the texture coordinates values
- The color retrieved from the texture map should be used for the material's diffuse color in your Phong shading calculation in your fragment shader

Suggestions for HW8

For 3D procedural texturing

- I would suggest that your mapping from 3D Cartesian space to RGB color space be based on a location specified in world, NOT camera, coordinates.
- So you will need to pass the location of the vertices in world coordinates to the fragment shader

Suggestions for HW8

- Write your procedural texture function in your fragment shader based on the interpolated value of the vertex location
- The color computed by your procedure should be used for the material's diffuse color in your Phong shading calculation in your fragment shader
- For the 3D procedural texturing you will only need to change your vertex and fragment shaders. I don't think that you will need to change any of your HW6 application code