Objectives

- Introduce Mapping Methods
  - Texture Mapping
  - Environment Mapping
  - Bump Mapping

- Consider basic strategies
  - Forward vs backward mapping
  - Point sampling vs area averaging

Texture Mapping

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

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

- Where does mapping take place?
  - Mapping techniques are implemented at the end of the rendering pipeline
    - Very efficient because few polygons make it past the clipper

Bump Mapping

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

Coordinate Systems

- 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
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 a 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

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
  \[ x = r \cos 2\pi u \]
  \[ y = r \sin 2\mu \]
  \[ z = v/h \]
- maps rectangle in \( u, v \) 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
  \[ x = r \cos 2\pi u \]
  \[ y = r \sin 2\mu \cos 2\pi v \]
  \[ z = r \sin 2\mu \sin 2\pi v \]
- 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

Area Averaging

A better but slower option is to use area averaging

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

x y z
image
display

Geometry

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

vertices
image

Fragment processor
texel pipeline
gometry pipeline

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
• WebGL1 supports only 2 dimensional texture maps
  - No need to enable as in desktop OpenGL
  - Desktop OpenGL supports 1-4 dimensional texture maps
  - Dimensions should be power of 2 for mip-mapping
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

```
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)];
```

```
function quad(a, b, c, d) {
  pointsArray.push(vertices[a]);
  colorsArray.push(vertexColors[a]);
  texCoordsArray.push(texCoord[0]);
  pointsArray.push(vertices[b]);
  colorsArray.push(vertexColors[a]);
  texCoordsArray.push(texCoord[1]);
  // etc
}
```

Mapping a Texture

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

```plaintext
Texture Space

Object Space

(s, t) = (0.2, 0.8)
```

Interpolation

WebGL uses interpolation to find proper texels from specified texture coordinates. Can be distortions due to a poor selection of texture coordinates.
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

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);
```

Filter Modes

- Modes determined by
  `gl.texParameteri(target, type, mode)`

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
```

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
  `gl.texImage2D(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);`
- Have WebGL make your mipmap with
  `gl.generateMipmap();`
*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

*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

*Applying Textures*

- Texture can be applied in many ways
  - Texture fully determines color
  - Modulated with a computed color
  - Blended with an environmental color
- Fixed function pipeline has a function `glTexEnv` to set mode
  - Deprecated
  - Can get all desired functionality via fragment shader
- Can also use multiple texture units

*Typical Code*

```javascript
// 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 );
```
Applying Textures

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

```glsl
varying vec4 color; // color from rasterizer
varying vec2 texCoord; // texture coordinate from rasterizer
uniform sampler2D texture; // texture object from application

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

Texture Object

```glsl
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, 2, 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

```glsl
var vTexCoord = gl.getAttribLocation(program, "vTexCoord");
gl.enableVertexAttribArray(vTexCoord);
gl.vertexAttribPointer(vTexCoord, 2, gl.FLOAT, false, 0, 0);

// 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(gl.getUniformLocation(program, "texture"), 0);
```

Reflection and Environment Maps

```glsl
var vTexCoord = gl.getAttribLocation(program, "vTexCoord");
gl.enableVertexAttribArray(vTexCoord);
gl.vertexAttribPointer(vTexCoord, 2, gl.FLOAT, false, 0, 0);

// 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(gl.getUniformLocation(program, "texture"), 0);
```

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**

- 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

**Mapping to a Sphere**

- Use largest magnitude component of $R$ to determine face of cube.
- Other two components give texture coordinates.

**Hemisphere Map as a Texture**

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

- WebGL1 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
  ```
  vec4 texColor = textureCube(mycube, texcoord);
  ```
- Texture coordinates must be 3D
  - usually are given by the vertex location so we don't need to 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 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

vs Cube Image
Doing it in WebGL

```javascript
// Create texture object from six images
textureMap = gl.createTexture();
for (var i = 0; i < 6; i++) {
  varamplerCube = i.toString();
  gl.bindTexture(gl.TEXTURE_CUBE_MAP_POSITIVE_X, 0, gl.RGBA, gl.UNSIGNED_BYTE, image1);
  gl.bindTexture(gl.TEXTURE_CUBE_MAP_POSITIVE_Y, 0, gl.RGBA, gl.UNSIGNED_BYTE, image2);
  gl.bindTexture(gl.TEXTURE_CUBE_MAP_POSITIVE_Z, 0, gl.RGBA, gl.UNSIGNED_BYTE, image3);
  gl.bindTexture(gl.TEXTURE_CUBE_MAP_NEGATIVE_X, 0, gl.RGBA, gl.UNSIGNED_BYTE, image4);
  gl.bindTexture(gl.TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, gl.RGBA, gl.UNSIGNED_BYTE, image5);
  gl.bindTexture(gl.TEXTURE_CUBE_MAP_NEGATIVE_Z, 0, gl.RGBA, gl.UNSIGNED_BYTE, image6);
  gl.activeTexture(gl.TEXTURE0);
}
```
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
Tangent Plane

\[ n \]

\[ p_u \]

\[ p_v \]

Equations

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

\[ p_u = [\partial x/\partial u, \partial y/\partial u, \partial z/\partial u]^T \]

\[ p_v = [\partial x/\partial v, \partial y/\partial v, \partial z/\partial v]^T \]

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

Displacement Function

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

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

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

Perturbed Normal

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

\[ p'_u = p_u + (\partial d/\partial u)n + d(u,v)n_u \]

\[ p'_v = p_v + (\partial d/\partial v)n + d(u,v)n_v \]

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

Approximating the Normal

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

\[ \approx n + (\partial d/\partial u)n \times p_v + (\partial d/\partial v)n \times p_u \]

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 \( \partial d/\partial u \) and \( \partial d/\partial v \)

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 \( D = [d_i] \)
- Then \( \partial d/\partial u = d_i - d_{i+1} \)
  and \( \partial d/\partial v = d_i - d_{i+1} \)
- **Embossing**: multipass approach using floating point buffer
Example

Single Polygon and a Rotating Light Source

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