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

game model  texture mapped

Environment Mapping

Bump 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

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

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

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

\[ x = r \cos \ 2\pi \ u \]
\[ y = r \sin \ 2\pi \ u \cos \ 2\pi \ v \]
\[ z = r \sin \ 2\pi \ u \sin \ 2\pi \ v \]
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
  - Miss blue stripes
- Point samples in texture space

Area Averaging

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

Go to ART Texture Mapping slides

WebGL Texture Mapping
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 only supports 2 dimensional texture maps
  - Dimensions should be power of 2 for mip-mapping
- These restrictions removed in WebGL 2, and now supports many texture types
- We’ll still focus on 2D textures
Define Image as a Texture

```javascript
// Define Image as a Texture

// Initialize new Uint8Array with texture size and RGBA format
var image1 = new Uint8Array(4*texSize*texSize);

// Iterate over texture coordinates and set pixel values
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;
    }
}
```

A Checkerboard Image

```javascript
// A Checkerboard Image

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;
    }
}
```

Loading a Texture Image

```javascript
// Loading a Texture Image

function loadTexture(url) {
    let texture = gl.createTexture();
    let image = new Image();
    // Put in placeholder image data until actual image loads.
    configureTexture(texture, null);

    image.addEventListener("error", (event) => {
        console.log(`error evt: ${JSON.stringify(event)}`);
    });

    image.addEventListener("load", (event) => {
        configureTexture(texture, image);
        window.requestAnimationFrame(render);
    });

    image.crossOrigin = "Anonymous";
    image.src = url + ":dont-use-cache";
    return texture;
}
```

Mapping a Texture

```javascript
// Mapping a Texture

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[b]);
    texCoordsArray.push(texCoord[1]);
}
```

Cube Example

```javascript
// Cube Example

var texCoord = {
    vec2(0, 0),
    vec2(0, 1),
    vec2(1, 1),
    vec2(1, 0)
};
```

Interpolation

```javascript
// 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 per 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

```glsl```
```gl
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP)
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT)
```
```// End of glsl```

Filter Modes

Modes determined by

```glsl```
```gl
  gl.texParameteri( target, type, mode )
```
```// End of glsl```
Texture Object I

```javascript
function configureTexture(texture, image) {
  gl.bindTexture(gl.TEXTURE_2D, texture);
  if (image) {
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
    gl.generateMipmap(gl.TEXTURE_2D);  
  } else {
    let pixel = new Uint8Array([0, 0, 150, 255]);  // dark blue pixel
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, 1, 1, 0, gl.RGBA, gl.UNSIGNED_BYTE, pixel);
  }
}
```

Texture Object II

```javascript
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.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT);
gl.activeTexture(gl.TEXTURE0);
gl.uniform1i(gl.getUniformLocation(program, "utexture"), 0);
```

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

```javascript
var aPosition = gl.getAttribLocation(program, "aPosition");
gl.enableVertexAttribArray(aPosition);
gl.vertexAttribPointer(aPosition, 4, gl.FLOAT, false, 0, 0);

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

```javascript
var aTextureCoord = gl.getAttribLocation(program, "aTexCoord");
gl.enableVertexAttribArray(aTextureCoord);
gl.vertexAttribPointer(aTextureCoord, 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 or normal if needed

```glsl
in vec4 aPosition; //vertex position in object coordinates
in vec4 aColor;    //vertex color from application
in vec2 aTexCoord; //texture coordinate from application
out vec4 vcolor;   //output color to be interpolated
out vec2 vtexCoord; //output tex coordinate to be interpolated
```

Applying Textures

- Texture can be applied in many ways
  - Texture fully determines color
  - Modulates a computed color
  - Used as diffuse color during Phong shading
- 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

```glsl```
```
Applying Textures

- Textures are applied during fragments shading by a sampler
- Samplers return a color from a texture object
- This example modulates vcolor from rasterizer

```glsl
in vec4 vcolor;    // color from rasterizer
in vec2 vtexCoord; // texture coordinate from rasterizer
uniform sampler2D utexture; // texture object from application
out vec4 fcolor;   // final color

void main() {
    fcolor = vcolor * texture( utexture, vtexCoord );
}
```

Reflection and Environment Maps

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

Hemisphere Map as a Texture

• 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

Cube Map

Indexing into Cube Map

• 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**

- 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 (samplerCube)
  ```
  vec4 texColor = texture(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 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

**vs Cube Image**
Doing it in WebGL

```javascript
// WebGL code for texture mapping

gl.textureMap2D(  
  gl.TEXTURE_CUBE_MAP_POSITIVE_X,  
  level, rows, columns, border, gl.RGBA,  
  gl.UNSIGNED_BYTE, image1)

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

Example

- 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)
  - Each face of room can be a texture of one texel
    ```javascript
    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
// Texture object code

cubeMap = gl.createTexture();
gl.bindTexture(gl.TEXTURE_CUBE_MAP, cubeMap);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_POSITIVE_X, 0, gl.RGBA,  
  1, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, red);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_NEGATIVE_X, 0, gl.RGBA,  
  1, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, green);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_POSITIVE_Y, 0, gl.RGBA,  
  1, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, blue);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, gl.RGBA,  
  1, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, cyan);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_POSITIVE_Z, 0, gl.RGBA,  
  1, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, yellow);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_NEGATIVE_Z, 0, gl.RGBA,  
  1, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, magenta);
gl.activeTexture(gl.TEXTURE0);
gl.uniform1i(gl.getUniformLocation(program, "texMap"), 0);
```

Vertex Shader

```javascript
// Vertex shader code

in vec4 aPosition;
in vec4 aNormal;
out vec3 vR;
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*ry*rx;
  vec4 eyePos = ModelViewMatrix*aPosition;
  vec4 N = ModelViewMatrix*aNormal;
  vR = reflect(eyePos.xyz, N.xyz);
}
```

Fragment Shader

```javascript
// Fragment shader code

precision mediump float;
in vec3 vR;
out vec4 fColor;
uniform samplerCube texMap;
void main(){
  vec3 R = normalize(vR);
  vec4 texColor = texture(texMap, R);
fColor = texColor;
}
```

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
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
Tangent Plane

\[ p_{u} = \left[ \frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u} \right]^T \]
\[ p_{v} = \left[ \frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v} \right]^T \]
\[ n = \frac{p_{u} \times p_{v}}{\| p_{u} \times p_{v} \|} \]

Equations

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

\[ n' = p'_{u} \times p'_{v} \]
\[ \approx n + (\partial d/\partial u) n \times p_{u} + (\partial d/\partial v) n \times p_{v} \]
The vectors \( n \times p_{u} \) and \( n \times p_{v} \) 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 \( D = [d_{i,j}] \)
• Then \( \partial d/\partial u \approx d_{i,j} - d_{i,j+1} \)
• \( \partial d/\partial v \approx d_{i,j} - d_{i+1,j} \)
• Embossing: multipass approach using floating point buffer
Example

Single Polygon and a Rotating Light Source

Angel and Shreiner: Interactive Computer Graphics 7E © Addison-Wesley 2015

Biparametric Patch

• (u,v) pair maps to a 3D point on patch
• Can use the same (u,v) for texture lookup

Angel and Shreiner: Interactive Computer Graphics 6E © Addison-Wesley 2012

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 34, 38, 47, 48, 52, 53, 55 to define and access 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 can 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
• You will need to send the light's diffuse color via a uniform variable, instead of diffuseProduct

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

Shader Code

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
• RGB colors should be between 0 and 1
• For the 3D procedural texturing you will need to change your vertex and fragment shaders. Very little of your HW6 application code will need to change (except sending the light diffuse color)