Texture Mapping

CS 537 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

generic 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

Mapping Functions

- Basic problem is how to find the maps
- Consider mapping from texture coordinates to a point on 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

\[
\begin{align*}
x &= r \cos 2\pi u \\
y &= r \sin 2\pi u \cos 2\pi v \\
z &= r \sin 2\pi u \sin 2\pi v
\end{align*}
\]

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

miss blue stripes

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

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
OpenGL Texture Mapping

CS 537 Interactive Computer Graphics
Prof. David E. Breen
Department of Computer Science
Objectives

• Introduce the OpenGL texture functions and options
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.
• Images and geometry flow through separate pipelines that join during fragment processing
  - “complex” textures do not affect geometric complexity

vertices → geometry pipeline

image → pixel pipeline

fragment processor
Specify a Texture Image

• Define a texture image from an array of texels (texture elements) in CPU memory
  
  ```glubyte my_texels[512][512][3];```

• Define as any other pixel map
  - Scanned image
  - Generate by application code

• Enable texture mapping
  - `glEnable(GL_TEXTURE_2D)`
  - OpenGL supports 1-4 dimensional texture maps
Define Image as a Texture

```c
glTexImage2D( 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: how texel is stored
w, h: width and height of texels in pixels
border: used for smoothing (discussed later)
format and type: describe texels
texels: pointer to texel array

```c
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 512, 512, 
    0, GL_RGB, GL_UNSIGNED_BYTE, my_texels); 
```
Mapping a Texture

• Application or shaders define texture coordinates
• Texture coordinates specified at each vertex

Texture Space

Object Space

(s, t) = (0.2, 0.8)

(0.4, 0.2)

(0.8, 0.4)
Typical Code

• Application sending texture coordinates

```c
offset = 0;
GLuint vPosition = glGetAttribLocation( program, "vPosition" );
 glEnableVertexAttribArray( vPosition );
 glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(offset) );

offset += sizeof(points);
GLuint vTexCoord = glGetAttribLocation( program, "vTexCoord" );
 glEnableVertexAttribArray( vTexCoord );
 glVertexAttribPointer( vTexCoord, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(offset) );
```
Interpolation

OpenGL 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
Texture Parameters

• OpenGL 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

\[
\begin{align*}
glTexParameteri( \text{GL_TEXTURE_2D}, & \text{GL_TEXTURE_WRAP_S}, \text{GL_CLAMP} ) \\
glTexParameteri( \text{GL_TEXTURE_2D}, & \text{GL_TEXTURE_WRAP_T}, \text{GL_REPEAT} )
\end{align*}
\]

\[
\begin{array}{ccc}
\text{texture} & \text{GL\_REPEAT} & \text{GL\_CLAMP} \\
\text{wrapping} & \text{wrapping}
\end{array}
\]
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.

Texture  Polygon  Magnification  

Texture  Polygon  Minification
Filter Modes

Modes determined by

\[-\text{glTexParameteri}(\text{target}, \text{type}, \text{mode})\]

\[
\text{glTexParameteri}(\text{GL\_TEXTURE\_2D}, \text{GL\_TEXTURE\_MAG\_FILTER}, \\
\text{GL\_NEAREST});
\]

\[
\text{glTexParameteri}(\text{GL\_TEXTURE\_2D}, \text{GL\_TEXTURE\_MIN\_FILTER}, \\
\text{GL\_LINEAR});
\]

Note that linear filtering requires a border of an extra texel for filtering at edges (\text{border} = 1)
Mipmapped Textures

- Mipmapping allows for prefILTERED texture maps of decreasing resolutions
- Lessens interpolation errors for smaller textured objects
- Declare mipmap level during texture definition
  \[
  \text{glTexImage2D( GL\_TEXTURE\_2D, level, ... )}
  \]
- Have OpenGL make your mipmap
  \[
  \text{glGenerateMipmap(target)}
  \]
- Mipmaps invoked by setting sampling
  \[
  \text{glTexParameteri(GL\_TEXTURE\_2D, GL\_TEXURE\_MIN\_FILTER, GL\_LINEAR\_MIPMAP\_LINEAR)};
  \]
Example

- Point sampling
- Linear filtering

- Mipmapped point sampling
- Mipmapped linear filtering
Using Texture Objects

1. specify textures in texture objects
2. set texture filter
3. set texture wrap mode
4. bind texture object
5. enable texturing
6. supply texture coordinates for vertex
   - coordinates can also be generated
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 OpenGL
  - 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
GLubyte image[64][64][3];

// Create a 64 x 64 checkerboard pattern
for ( int i = 0; i < 64; i++ ) {
    for ( int j = 0; j < 64; j++ ) {
        GLubyte c = (((i & 0x8) == 0) ^ ((j & 0x8) == 0)) * 255;
        image[i][j][0] = c;
        image[i][j][1] = c;
        image[i][j][2] = c;
    }
}
Adding Texture Coordinates

```c
void quad( int a, int b, int c, int d )
{
    quad_colors[Index] = colors[a];
    points[Index] = vertices[a];
    tex_coords[Index] = vec2( 0.0, 0.0 );
    index++;
    quad_colors[Index] = colors[a];
    points[Index] = vertices[b];
    tex_coords[Index] = vec2( 0.0, 1.0 );
    Index++;

    // other vertices
}
```

GLuint textures[1];
glGenTextures( 1, textures );

glBindTexture( GL_TEXTURE_2D, textures[0] );
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, TextureSize, TextureSize, 0, GL_RGB, GL_UNSIGNED_BYTE, image );
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 );
glActiveTexture( GL_TEXTURE0 );
GLuint vTexCoord = glGetAttribLocation( program, "vTexCoord" );
glEnableVertexAttribArray( vTexCoord );
glVertexAttribPointer( vTexCoord, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(offset) );

// 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
// glActiveTexture().
glUniform1i( glGetUniformLocation(program, "texture"), 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

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

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

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

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

void main()  {
    gl_FragColor = color * texture2D( texture, texCoord );
}
```
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
Suggestions for HW9

• 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 HW9

• 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 HW9

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 HW9

• 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