Topics

Texture mapping in OpenGL
Mapping

Modify color in fragment processing after rasterization

Three Major Mapping Methods

- **Texture Mapping**
  - Uses images to fill inside of polygons

- **Environment (reflection mapping)**
  - Uses a picture of the environment for texture maps of reflection surface
  - Allows simulation of highly specular surfaces

- **Bump mapping**
  - Emulates altering normal vectors during the rendering process
Texture Mapping

Textures are stored in images - 2D arrays.

Each element is called a texel

The idea is simple---map an image to a surface or map every texel to a point on a geometric object

However, there are 3 or 4 coordinate systems involved

Texels: texture elements

2D image

3D surface
Texture Mapping

parametric coordinates

texture coordinates

world coordinates

window coordinates

\[ T(s, t) \]
Backward Mapping

We really want to go backwards

- Given a pixel, we want to know to which point on an object it corresponds, the preimage (inverse) of a pixel
- Given a point on an object, we want to know to which point in the texture it corresponds

Backward mapping

\[ s = s(x, y, z, w) \]
\[ t = t(x, y, z, w) \]

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, e.g., a sphere, cylinder, or cube.

First mapping can be:

- Cylindrical mapping
- Spherical mapping
- Box mapping
Second Mapping: From Intermediate Object to Actual Object

Three strategies:

• from intermediate, along normal of intermediate until intersect with the object
• from object, along normal of the object until intersect with the intermediate
• Vectors from center of intermediate/object, intersect the object and the intermediate
Aliasing

Backward mapping for the centers of pixels

Point sampling of the texture/object can lead to aliasing errors

miss blue stripes

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

A better but slower option is to use *area averaging* of the texture map over the preimage.

Cannot handle high-frequency components, e.g., the stripe pattern – sampling at higher frequencies.
OpenGL Texture Mapping

Texture mapping is part of fragment processing

Three steps to applying a texture

1. Generate the texture map
   - read or generate image
   - assign to texture
   - enable texturing
2. assign texture coordinates to vertices
   - Texture coordinates can be interpolated
   - Proper mapping function is left to application
3. specify texture parameters
   - wrapping, filtering
Texture Mapping

geometry

display

image

S t x y z
Texture Mapping and the OpenGL Pipeline

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

[Diagram showing the pipeline flows from vertices through the geometry pipeline, then to the fragment processor, and from images through the pixel pipeline to the fragment processor.]
Shader-based OpenGL Texture Mapping

- Create a *texture object* and load texel data into the texture object
- Associate texture coordinates with vertices of object
- Associate a *texture sampler* with each texture map
- Get the texel values from the texture sampler from shader
Shader-based OpenGL Texture Mapping

- Create a *texture object* and load texel data into the texture object
  - Associate texture coordinates with vertices of object
  - Associate a *texture sampler* with each texture map
  - Get the texel values from the texture sampler from shader
Create Texture Objects

GLuint textures[# of texture objects];

• Create texture objects and return n unused names for texture objects

void glGenTextures(GLsizei n, GLuint *textures);

Number of texture objects
Bind Texture Objects

```c
void glBindTexture(GLenum target, GLuint texture);
```

- Type and dimensionality of target, e.g., `GL_TEXTURE_2D`
- Create a new texture object with the assigned name
- Bind an existing texture object and make it active.
Select Texture Units

In OpenGL, textures are bound to OpenGL context by *texture units*.

The context can support multiple texture units – many textures can be bound to the same context at the same time.

Select Texture Units

Step 1: select the (first) texture unit by

```c
void glActiveTexture(GLenum texture);
```

A constant of the form `GL_TEXTUREi`, `i ∈ [0, max # of texture objects − 1]`

If there is only one texture unit, use `GL_TEXTURE0`

Step 2: bind the texture object

```c
void glBindTexture(GLenum target, GLuint texture);
```

For example,

```c
    glActiveTexture( GL_TEXTURE0 );
    glBindTexture( GL_TEXTURE_2D, textures[0] );
```
Define a texture image from an array of texels (texture elements) in CPU memory

- Generate by application code
- Load from scanned image

```
Glubyte my_texels[height][width][channel];
```
Example: Checkerboard Texture

GLubyte my_texels [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;
    }
}
Define Image as a Texture

`glTexImage2D( target, level, internal format, w, h, border, format, type, texels );`

- **target**: type of texture, e.g. `GL_TEXTURE_2D`
- **level**: used for mipmapping (discussed later), e.g., 0 w/o mipmap
- **Internal format**: format to store texture data, e.g., `GL_RGB` or `GL_RGBA`
- **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

For example,

`glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 512, 512, 0, GL_RGB, GL_UNSIGNED_BYTE, my_texels);`
Shader-based OpenGL Texture Mapping

• Create a *texture object* and load texel data into the texture object

• Associate texture coordinates with vertices of object

• Associate a *texture sampler* with each texture map

• Get the texel values from the texture sampler from shader
In the 512x512 texture space, (0,0) corresponds to my_texels(0,0), and (1.0,1.0) corresponds to my_texels(511,511)
vec2    tex_coords[NumVertices];
void quad( int a, int b, int c, int d )
{
    points[Index] = vertices[a];
    tex_coords[Index] = vec2( 0.0, 0.0 );
    index++;

    points[Index] = vertices[b];
    tex_coords[Index] = vec2( 0.0, 1.0 );
    Index++;

    // other vertices
}
Associate Texture Coordinates with Vertices of Object

Treat texture coordinates as a vertex attribute, similar to vertices and vertex colors.

Pass the vertex texture coordinates to the vertex shader.

The rasterizer interpolates the vertex texture coordinates to fragment texture coordinates.
Interpolation

OpenGL uses interpolation to find proper texels from specified texture coordinates

Can be distortions

good selection of texture coordinates

poor selection of texture coordinates
Angel’s Example Code

// Create a vertex array object

GLuint vao;

glGenVertexArrays( 1, &vao );

glBindVertexArray( vao );

// Create and initialize a buffer object

GLuint buffer;

glGenBuffers( 1, &buffer );

glBindBuffer( GL_ARRAY_BUFFER, buffer );

glBufferData( GL_ARRAY_BUFFER, sizeof(vertices) + sizeof(tex_coords), NULL, GL_STATIC_DRAW );
Angel’s Example Code

GLintptr offset = 0;

glBufferSubData( GL_ARRAY_BUFFER, offset, sizeof(vertices), vertices );

offset += sizeof(vertices);

glBufferSubData( GL_ARRAY_BUFFER, offset, sizeof(tex_coords), tex_coords );
Angel’s Example Code

// after load shaders, use the program

offset = 0;

GLuint vPosition = glGetUniformLocation( program, "vPosition" );

glEnableVertexAttribArray( vPosition );

glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,BUFFER_OFFSET(offset) );

offset += sizeof(vertices);

GLuint vTexCoord = glGetUniformLocation( program, "vTexCoord" );

glEnableVertexAttribArray( vTexCoord );

glVertexAttribPointer( vTexCoord, 2,GL_FLOAT,GL_FALSE, 0,BUFFER_OFFSET(offset) );
Shader-based OpenGL Texture Mapping

• Create a *texture object* and load texel data into the texture object

• Associate texture coordinates with vertices of object

• Associate a *texture sampler* with each texture map

• Get the texel values from the texture sampler from shader
Applying Textures

Textures are read/applied during fragments shading by a **sampler**, often in a fragment shader.

A texture object has a built-in sampler object.

Samplers return a texture color from a texture object.

Set the value of the fragment shader texture sampler variable to the appropriate texture unit selected by calling `glActiveTexture()`.

```
glUniform1i( glGetUniformLocation(program, "tex"), 0 );
```

The selected texture unit

Fragment shader texture sampler variable
Shader-based OpenGL Texture Mapping

- Create a *texture object* and load texel data into the texture object

- Associate texture coordinates with vertices of object

- Associate a *texture sampler* with each texture map

- Get the texel values from the texture sampler from shader
Applying Textures: Vertex Shader

Usually vertex shader will output texture coordinates to be rasterized

```cpp
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
void main()
{
    color  = vColor;
    texCoord    = vTexCoord;
    gl_Position = vPosition;
}
```
Applying Textures in Fragment Shader

in vec4 color;  //color from rasterizer

in vec2 texCoord; //texture coordinate from rasterizer

uniform sampler2D tex; //texture object from application

Type of sampler, e.g., sampler3D, samplerCube

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

Get texels from the sampler
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
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
Setting Texture Parameters

```c
glTexParameteri(GLenum target, GLenum pname, Type param);
```

Can be f, fv, iv, etc.

- **target**: type of texture, e.g. `GL_TEXTURE_2D`
- **pname**: the symbolic name of a single-valued texture parameter, e.g., `GL_TEXTURE_WRAP_S` and `GL_TEXTURE_MIN_FILTER`
  - Need to run this function for every parameter
- **param**: the value of parameter

OpenGL manual for `glTexParameteri`

Wrapping Mode

Valid range of texture coordinates: $0 \leq s, t \leq 1$

**Clamping mode:** if $s, t > 1$ use 1, if $s, t < 0$ use 0

• Avoid wrapping artifacts for nonrepeated texture patterns

**Wrapping mode:** use $s, t \mod 1$

• Preferred for repeated texture patterns, e.g., checkerboard

```c
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
```

GL_TEXTURE_WRAP_S(T): set texture coordinate $s (t)$

Parameters to choose: GL_CLAMP, GL_CLAMP_TO_EDGE, GL_CLAMP_TO_BORDER, GL_MIRRORED_REPEAT, GL_REPEAT, or GL_MIRROR_CLAMP_TO_EDGE.
Wrapping Mode

**Clamping mode:** if \( s, t > 1 \) use 1, if \( s, t < 0 \) use 0

**Wrapping mode:** use \( s, t \) modulo 1

Example: repeated texture patterns

![Texture Patterns](image)

- Texture
- GL_REPEAT wrapping
- GL_CLAMP wrapping

Wrapping Mode

Example: nonrepeated texture

https://open.gl/textures
Magnification and Minification

**Minification**: more than one texel can cover a pixel

**Magnification**: more than one pixel can cover a texel

Can use point sampling (nearest texel) or linear filtering (2 x 2 filter) to obtain texture values.
Magnification and Minification: Filter Modes

```c
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

GL_NEAREST is faster, but causes jitter edge.
GL_LINEAR is the default mode.

Note that linear filtering requires a border of an extra texel for filtering at edges (border = 1)