Lecture 19:
OpenGL Texture Mapping
Objectives

• Introduce the OpenGL texture functions and options
Basic Strategy

There are 3 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

display

geometry

image

x

y

z

t

s
Texture Example

• This is a 256 x 256 image.

• It can be used as a texture map in OpenGL.

• The texture has been mapped to a rectangular polygon which is viewed in perspective.
Texture Mapping and the OpenGL Pipeline

• Images and geometry flow through separate pipelines that join during fragment processing.

• Hence, “complex” textures do not affect geometric complexity.

vertices → geometry pipeline

image → pixel pipeline

fragment processor
Texture Resolution vs Geometry Resolution

- We can have different resolutions for texture and geometry
- Texture processing is not as complex as geometry processing
- High resolution textures give more realistic appearance
- High resolution texture mapped on low resolution geometry still looks good while being light on the graphics pipeline
- Ground textures in your Project are perfect examples of this
Specifying a Texture Image

• Define a texture image from an array of *texels* (*texture elements*) in CPU memory
  \[
  \text{GLubyte my\_texels[512][512]};
  \]

• Define as any other pixel map
  - Scanned image or camera image
  - Generated by application program

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

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

  where

  `target`: type of texture, e.g. `GL_TEXTURE_2D`
  `level`: used for **mipmapping** (discussed later)
  `components`: elements per texel
  `w, h`: width and height of `texels` in pixels
  `border`: used for smoothing (discussed later)
  `format, type`: the format and type of the texels
  `texels`: pointer to texel array

- Example:
  
  `glTexImage2D( GL_TEXTURE_2D, 0, 3, 512, 512, 0,
  GL_RGB, GL_UNSIGNED_BYTE, my_texels);`
// pass the vertex coordinates to vertex shader
offset = 0;
GLuint vPosition = glGetUniformLocation(program,"vPosition");

glEnableVertexAttribArray(vPosition);

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

// piggy-back the texture coordinates at the
// end of the buffer and pass it to vertex shader
offset += sizeof(points);

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

glEnableVertexAttribArray(vTexCoord);

glVertexAttribPointer(vTexCoord, 2, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(offset) );
Interpolation

• Defining texture coordinates for mapping textures onto rectangles is easy. How to define texture coordinates for complex objects can be tricky.

• OpenGL uses interpolation to find proper texels from specified texture coordinates. Distortions may result.
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
  - Texture sampling mode allows us to specify using area averaging instead of point samples
  - Mipmapping allows us to use textures at multiple resolutions
  - Environment parameters determine how texture mapping interacts with shading

- \texttt{glTexParameter*(GLenum target, GLenum pname, GLint value);} // * can be i or f
- \texttt{glTexParameter}i(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);
Wrapping Mode

- Want $s$ and $t$ in the range $0 \ldots 1$. Can use clamping or wrapping to force them in the $[0,1]$ range.

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

- **Wrapping**: use $s,t > 1$ then modulo 1

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

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

![Wrapping Mode Diagram](image)
Texture Sampling

- Aliasing in textures is a major problem. When we map texture coordinates to the texels array, we rarely get a point that is exactly at the centre of the texel.
- OpenGL supports the following options for sampling textures:
  1. **Point sampling** – use the value of the texel that is closest to the texture coordinates output by the rasterizer
  2. **Linear filtering** – use the weighted average of a group of texels in neighbourhood of the texture coordinates output by the rasterizer

For **point sampling**, the value of the bottom-left cyan texel would be used.

For **linear filtering**, the weighted average of the four cyan texels would be used as the texture value for the point sample.
Magnification and Minification

In deciding how to use the texel values to obtain the final texture value, the size of the pixel on the screen may be larger or smaller than the texel:

- The texel is smaller than one pixel (minification) or
- The texel is larger than one pixel (magnification)

One texel is smaller than one pixel

Minification

One texel is smaller than one pixel

Magnification
The filter mode can be specified by calling:

```
glTexParameteri( target, type, mode )
```

**Magnification:**

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

**Minification:**

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

For point sampling, replace GL_LINEAR by GL_NEAREST.

Note that linear filtering requires a border of an extra texel for filtering at edges (border = 1)
Magnification and Minification (cont.)

For objects that project to a smaller area on the screen, we don’t need to keep the original full resolution of the texel array.

OpenGL allows us to create a series of texture arrays at reduced sizes, e.g., for a $64 \times 64$ original array, we can set up $64 \times 64, 32 \times 32, 16 \times 16, 8 \times 8, 4 \times 4, 2 \times 2,$ and $1 \times 1$ arrays by calling:

```
glGenerateMipmap(GL_TEXTURE_2D);
```

One texel is smaller than one pixel

Minification

One texel is smaller than one pixel

Magnification
Mipmapped Textures

• **Mipmapping** (sometimes called MIP mapping) is a technique where an original high-resolution texture map is scaled and filtered into multiple resolutions within the texture file.

• **Mipmapping** allows for prefILTERED texture maps of decreasing resolutions.

• Reduces interpolation errors for smaller textured objects.

• You need to firstly declare the mipmap level during texture definition (by calling, e.g., `glTexImage2D`). Thus the order of function calls is (e.g.):

```c
glTexImage2D( GL_TEXTURE_2D, 0, ... );

glGenerateMipmap(GL_TEXTURE_2D);
```

Indicates that we want to keep all resolutions (all the way to base level 0, i.e. the original texel array).
Examples

point sampling

mipmapped point sampling

linear filtering

mipmapped linear filtering
Texture Functions

You can control how texture is applied by calling:

```c
GLuintenv{fi}[v](GL_TEXTURE_ENV, name, param)
```

// e.g. glTexEnvf, glTeEnvfv, glTexEnvi, glTexEnviv

**name** – the symbolic name of a texture environment parameter, e.g.,

```c
GL_TEXTURE_ENV_MODE
```

(there are 19,… see docs)

**param** – may be one of the following:
- **GL_MODULATE**: modulate with computed shade
- **GL_BLEND**: blend with an environmental color
- **GL_REPLACE**: use only texture color

Example: `glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_BLEND);`

sets the texture function to **GL_BLEND**
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)

See example1.cpp in the CHAPTER07_CODE folder
Applying Textures

- Textures are applied during fragment shading by a sampler

- Samplers return a texture colour from a texture object

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

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

```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 texture coordinate to be //interpolated
```
Checkerboard Texture

We can create our own texture map in the application. For example, creating a checkerboard texture:

```c
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) ^ // bitwise &, Hex, ^ XOR
            ((j & 0x8) == 0)) * 255;
        image[i][j][0] = c;
        image[i][j][1] = c;
        image[i][j][2] = c;
    }
}
```

Partial code from `example1.cpp` in the `CHAPTER07_CODE` folder
Adding Texture Coordinates

Code from an example in an earlier lecture:

```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
}
```

Partial code from example1.cpp in the CHAPTER07_CODE folder
Texture Object

```c
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 );

glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);

glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

glActiveTexture(GL_TEXTURE0);
```

Partial code from example1.cpp in the CHAPTER07_CODE folder
Texture Mapping Real Images

• If the texture to be mapped is a real image, it already has shading and shadows
  - Due to directional light sources
  - Due to self occlusions and occlusions from other objects

• These shading and shadows will not correspond to the light sources in your graphics world

• A good texture map should not include shading or shadows
  - Such an image can only be captured in controlled conditions with perfect diffused lighting i.e. light coming from all directions
  - Multiple large planar light sources can approximate diffused light
2D Texture on 3D Objects

• Wrapping 2D texture onto a 3D object is problematic, especially if the 3D shape is complex

• Unless the texture was on the 3D object already in the real world and the object was scanned

• And/Or some effort was put into unwrapping the texture
MakeHuman Texture Files

If we generate human with clothes option, we get texture files for the skin, hair, eyes, eyelashes, eyebrow, clothes, shoes, etc.
Further Reading


• Sec. 7.6 *Texture Mapping in OpenGL* (including all the subsections)