Fog example

- Fog is atmospheric effect
  - Better realism, helps determine distances
Fog

- Fog was part of OpenGL fixed function pipeline
- Programming fixed function fog
  - **Parameters:** Choose fog color, fog model
  - **Enable:** Turn it on
- Fixed function fog **deprecated**!!
- Shaders can implement even better fog
- **Shaders implementation:** fog applied in fragment shader just before display
Rendering Fog

- Mix some color of fog: $c_f$ + color of surface: $c_s$

$$c_p = f c_f + (1 - f) c_s \quad f \in [0,1]$$

- If $f = 0.25$, output color = 25% fog + 75% surface color

  - How to compute $f$?
  - 3 ways: linear, exponential, exponential-squared
  - Linear:

    $$f = \frac{Z_{end} - Z_p}{Z_{end} - Z_{start}}$$
Fog Shader Fragment Shader Example

float dist = abs(Position.z);
float fogFactor = (Fog.maxDist - dist)/
    Fog.maxDist - Fog.minDist);
fogFactor = clamp(fogFactor, 0.0, 1.0);
vec3 shadeColor = ambient + diffuse + specular
vec3 color = mix(Fog.color, shadeColor,fogFactor);
FragColor = vec4(color, 1.0);

\[ f = \frac{Z_{\text{end}} - Z_p}{Z_{\text{end}} - Z_{\text{start}}} \]

\[ c_p = fc_f + (1 - f)c_s \]
Fog

- Exponential \( f = e^{-d_f z_p} \)
- Squared exponential \( f = e^{-(d_f z_p)^2} \)
- Exponential derived from Beer’s law
  - **Beer’s law**: intensity of outgoing light diminishes exponentially with distance
Fog

- $f$ values for different depths ($z_p$) can be pre-computed and stored in a table on GPU.
- Distances used in $f$ calculations are planar.
- Can also use Euclidean distance from viewer or radial distance to create *radial fog*.
The Limits of Geometric Modeling

- Although graphics cards can render over 10 million polygons per second
- Many phenomena even more detailed
  - Clouds
  - Grass
  - Terrain
  - Skin
- Computationally inexpensive way to add details

Image complexity does not affect the complexity of geometry processing (transformation, clipping...)
Textures in Games

- Everything is a texture except foreground characters that require interaction
- Even details on foreground texture (e.g. clothes) is texture
Types of Texturing

1. geometric model
2. texture mapped
   Paste image (marble) onto polygon
Types of Texturing

3. Bump mapping
   Simulate surface roughness (dimples)

4. Environment mapping
   Picture of sky/environment over object
Texture Mapping

1. Define texture position on geometry
2. Projection
3. Texture lookup
4. Patch texel

3D geometry

2D projection of 3D geometry

2D image
Texture Representation

- Bitmap (pixel map) textures: images (jpg, bmp, etc) loaded
- Procedural textures: E.g. fractal picture generated in .cpp file
- Textures applied in shaders

Bitmap texture:

- 2D image - 2D array \( \text{texture}[\text{height}][\text{width}] \)
- Each element (or \text{texel} ) has coordinate \((s, t)\)
- \(s\) and \(t\) normalized to \([0,1]\) range
- Any \((s,t)\) => [red, green, blue] color
Texture Mapping

- Map? Each \((x,y,z)\) point on object, has corresponding \((s, t)\) point in texture
  
  \[
  s = s(x,y,z) \\
  t = t(x,y,z)
  \]

![Diagram](image_url)
6 Main Steps to Apply Texture

1. Create texture object
2. Specify the texture
   - Read or generate image
   - assign to texture (hardware) unit
   - enable texturing (turn on)
3. Assign texture (corners) to Object corners
4. Specify texture parameters
   - wrapping, filtering
5. Pass textures to shaders
6. Apply textures in shaders
Step 1: Create Texture Object

- OpenGL has **texture objects** (multiple objects possible)
  - 1 object stores 1 texture image + texture parameters
- First set up texture object

```c
GLuint mytex[1];
glGenTextures(1, mytex);                  // Get texture identifier
glBindTexture(GL_TEXTURE_2D, mytex[0]); // Form new texture object
```

- Subsequent texture functions use this object
- Another call to `glBindTexture` with new name starts new texture object
Step 2: Specifying a Texture Image

- Define input picture to paste onto geometry
- Define texture image as array of **texels** in CPU memory
  
  ```
  Glubyte my_texels[512][512][3];
  ```
- Read in scanned images (jpeg, png, bmp, etc files)
  
  - If uncompressed (e.g. bitmap): read into array from disk
  - If compressed (e.g. jpeg), use third party libraries (e.g. Qt, devil) to 
    uncompress + load

  bmp, jpeg, png, etc
Step 2: Specifying a Texture Image

- Procedural texture: generate pattern in application code
  
- Enable texture mapping
  - `glEnable(GL_TEXTURE_2D)`
  - OpenGL supports 1-4 dimensional texture maps
Specify Image as a Texture

Tell OpenGL: this image is a texture!!

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

- **target**: type of texture, e.g. `GL_TEXTURE_2D`
- **level**: used for mipmapping (0: highest resolution. More later)
- **components**: elements per texel
- **w, h**: width and height of `texels` in pixels
- **border**: used for smoothing (discussed later)
- **format, type**: describe texels
- **texels**: pointer to texel array

Example:

```c
glTexImage2D(GL_TEXTURE_2D, 0, 3, 512, 512, 0, GL_RGB, GL_UNSIGNED_BYTE, my_texels);
```
**Fix texture size**

- OpenGL textures must be power of 2
- If texture dimensions not power of 2, either
  1) Pad zeros
  2) Scale the Image

Remember to adjust target polygon corners – don’t want black texels in your final picture
6 Main Steps. Where are we?

1. Create texture object
2. Specify the texture
   - Read or generate image
   - assign to texture (hardware) unit
   - enable texturing (turn on)
3. Assign texture (corners) to Object corners
4. Specify texture parameters
   - wrapping, filtering
5. Pass textures to shaders
6. Apply textures in shaders
Step 3: Assign Object Corners to Texture Corners

- Each object corner \((x,y,z)\) => image corner \((s, t)\)
  - E.g. object \((200,348,100)\) => \((1,1)\) in image
- Programmer establishes this mapping
- Target polygon can be any size/shape
Step 3: Assigning Texture Coordinates

- After specifying corners, interior (s,t) ranges also mapped
- Example? Corners mapped below, abc subrange also mapped
Step 3: Code for Assigning Texture Coordinates

- **Example:** Trying to map a picture to a quad
- For each quad corner (vertex), specify
  - Specify vertex \((x,y,z)\),
  - Specify corresponding corner of texture \((s, t)\)
- May generate array of vertices + array of texture coordinates

\[
\text{points}[i] = \text{point3}(2,4,6); \\
\text{tex\_coord}[i] = \text{point2}(0.0, 1.0);
\]
Step 3: Code for Assigning Texture Coordinates

void quad( int a, int b, int c, int d )
{
    quad_colors[Index] = colors[a];            // specify vertex color
    points[Index] = vertices[a];                 // specify vertex position
    tex.coords[Index] = vec2( 0.0, 0.0 );  // specify corresponding texture corner
    index++;
    quad_colors[Index] = colors[b];
    points[Index] = vertices[b];
    tex.coords[Index] = vec2( 0.0, 1.0 ); // specify corresponding texture corner
    index++;

    // other vertices
}

points array
tex_coord array
colors array
Step 5: Passing Texture to Shader

- Pass vertex, texture coordinate data as vertex array
- Set texture unit

```c
offset = 0;
GLuint vPosition = glGetAttribLocation(program, "vPosition");
gleEnableVertexAttribArray(vPosition);
 glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0,BUFFER_OFFSET(offset));
offset += sizeof(points);
GLuint vTexCoord = glGetAttribLocation(program, "vTexCoord");
gleEnableVertexAttribArray(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.

glUniform1i(glGetUniformLocation(program, "texture"), 0);
```
Step 6: Apply Texture in Shader (Vertex Shader)

- Vertex shader receives data, output texture coordinates to fragment shader

```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

texCoord = vTexCoord
color = vColor
gl_Position = modelview * projection * vPosition
```
Step 6: Apply Texture in Shader (Fragment Shader)

- Textures applied in fragment shader
- 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 from application

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

- Output color of fragment
- Original color of object
- Lookup color of `texCoord(s,t)` in texture
Map textures to surfaces

- Texture mapping is performed in rasterization

- For each pixel, its texture coordinates \((s, t)\) interpolated based corners’ texture coordinates (why not just interpolate the color?)

- The interpolated texture \((s,t)\) coordinates are then used to perform texture lookup
Texture Mapping and the OpenGL Pipeline

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

Diagram:
- Vertices → geometry pipeline → Fragment processor
- Image → pixel pipeline → Fragment processor
6 Main Steps to Apply Texture

1. Create texture object
2. Specify the texture
   - Read or generate image
   - assign to texture (hardware) unit
   - enable texturing (turn on)
3. Assign texture (corners) to Object corners
4. Specify texture parameters
   - wrapping, filtering
5. Pass textures to shaders
6. Apply textures in shaders

still haven’t talked about setting texture parameters
References