Chapters VIII
Image Texturing
Where are We?

- While the vertex shader outputs the clip-space vertices, the rasterizer outputs a set of fragments at the screen space.

- The per-fragment attributes may include a normal vector, a set of texture coordinates, etc.

- Using these data, the fragment shader determines the final color of each fragment.

- Two most important things to do in the fragment shader
  - Lighting
  - Texturing

- Before moving on to the fragment shader, let’s see the basics of texturing.
Texture Coordinates

- The simplest among the various texturing methods is *image texturing*.
- An image texture is a 2D array of *texels* (texture elements). A texel’s location can be described by its center's coordinates.

For each fragment \( p \), we use its texture coordinates \((s,t)\) to find a location \( q \) in the texture such that the color at \( q \) can be retrieved and then applied to \( p \).
Texture Coordinates (cont’d)

- Scan conversion is done with the texture coordinates.

- GL multiplies the texture coordinates by the texture image resolution, $r_x$ and $r_y$.

\[
s' = s \times r_x \\
t' = t \times r_y
\]
Texture Coordinates (cont’d)

- It is customary to normalize the texture coordinates such that $s$ and $t$ range from 0 to 1. The texture coordinates can be freely plugged into different textures.
Multiple images of different resolutions can be glued to a surface without changing the texture coordinates.
Surface Parameterization

- The texture coordinates are assigned to the vertices of the polygon mesh. This process is called *surface parameterization* or simply *parameterization*.
- In general, parameterization requires unfolding a 3D surface onto a 2D planar domain.
Chart and Atlas

- The surface to be textured is often subdivided into a (hopefully small) number of patches.
- Each patch is unfolded and parameterized. Then, the artist draws an image for each patch. An image for a patch is often called a chart.
- Multiple charts are usually packed and arranged in a texture, which is often called an atlas.
**Texture Wrapping Mode**

- The texture coordinates \((s,t)\) are not necessarily in the range of \([0,1]\). The *texture wrapping mode* handles \((s,t)\) outside the range.
  - **Clamp-to-Edge**: The out-of-range coordinates are rendered with the edge colors.
  - **Repeat**: The texture is tiled at every integer junction.
  - **Mirror**: The texture is mirrored or reflected at every integer junction. We have smooth transition at the boundaries.

![Texture Wrapping Mode Diagram](image)
Texturing in GL

- Given an image file to be used as a texture, it is first of all loaded into the GL program. Assume that we use `TexData` to describe the structure of the loaded texture and `newTex` is a variable of `TexData`.

```c
struct TexData
{
    GLubyte* data;
    GLuint width;
    GLuint height;
    GLint format;
};

TexData newTex;

GLuint oTextures;
glGenTextures(1, &oTextures);
glBindTexture(GL_TEXTURE_2D, oTextures);
glTexImage2D(GL_TEXTURE_2D, 0, newTex.format, newTex.width, newTex.height,
0, newTex.format, GL_UNSIGNED_BYTE, newTex.pixels);
```

- Observe that the above is similar to the process of creating buffer objects for vertex array.
Texturing in GL

- Given an image file to be used as a texture, it is first of all loaded into the GL program. Assume that we use `TexData` to describe the structure of the loaded texture and `newTex` is a variable of `TexData`.

```c
struct TexData
{
    GLubyte* data;
    GLuint width;
    GLuint height;
    GLint format;
};

TexData newTex;

GLuint oTextures;
glGenTextures(1, &oTextures);
glBindTexture(GL_TEXTURE_2D, oTextures);
gTexImage2D(GL_TEXTURE_2D, 0, newTex.format, newTex.width, newTex.height, 0, newTex.format, GL_UNSIGNED_BYTE, newTex.pixels);
```

- Observe that the above is similar to the process of creating buffer objects for vertex array.
void glTexImage2D(GLenum target,
  GLint level,
  GLint internalFormat,
  GLsizei width,
  GLsizei height,
  GLint border,
  GLenum format,
  GLenum type,
  const GLvoid * data);

<table>
<thead>
<tr>
<th>target</th>
<th>GL_TEXTURE_2D for 2D image texture</th>
</tr>
</thead>
<tbody>
<tr>
<td>level</td>
<td>the mipmap level (later on this)</td>
</tr>
<tr>
<td>internalFormat</td>
<td>the number of color components such as GL_RGBA</td>
</tr>
<tr>
<td>width</td>
<td>the width of the texture image</td>
</tr>
<tr>
<td>height</td>
<td>the height of the texture image</td>
</tr>
<tr>
<td>border</td>
<td>OpenGL’s legacy argument that must be 0.</td>
</tr>
<tr>
<td>format</td>
<td>the pixel format such as GL_RGBA</td>
</tr>
<tr>
<td>type</td>
<td>the pixel data type such as GL_UNSIGNED_BYTE</td>
</tr>
<tr>
<td>data</td>
<td>the pointer to the image data in memory</td>
</tr>
</tbody>
</table>
Texture Wrapping Mode

- The texture coordinates \((s,t)\) are not necessarily in the range of \([0,1]\). The texture wrapping mode handles \((s,t)\) that are outside the range.
  - **Clamp-to-Edge**: The out-of-range coordinates are rendered with the edge colors.
  - **Repeat**: The texture is tiled at every integer junction.
  - **Mirrored-Repeat**: The texture is mirrored or reflected at every integer junction. We have smooth transition at the boundaries.
Texture Wrapping Mode (cont’d)

```c
void glTexParameteri(GLenum target,
                     GLenum pname,
                     GLint param);
```

- **target** target is `GL_TEXTURE_2D` for 2D image texture
- **pname** either `GL_TEXTURE_WRAP_S` or `GL_TEXTURE_WRAP_T` for texture wrapping
- **param** either `GL_CLAMP_TO_EDGE`, `GL_REPEAT`, or `GL_MIRRORED_REPEAT`

```c
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);
```
Consider a quad. For each fragment located at \((x, y)\) in the screen, its texture coordinates \((s, t)\) are *projected* onto \((s', t')\) in the texture space.

- Note that \((s', t')\) are floating-point values in almost all cases.
- Consequently, texels around \((s', t')\) are sampled. This sampling process is called *texture filtering*.
- The screen-space quad may appear larger than the image texture, and therefore the texture is *magnified* so as to fit to the quad. There are more pixels than texels.
In contrast, the screen-space quad may appear smaller than the image texture, and the texture is minified. The pixels are *sparsely projected* onto the texture space.

- **Summary**
  - Magnification: more pixels than texels
  - Minification: less pixels than texels
Filtering for Magnification

- Option 1: Nearest point sampling
  - A block of pixels can be mapped to a single texel.
  - Consequently, adjacent pixel blocks can change abruptly from one texel to the next texel, and a blocky image is often produced.
Option 2: Bilinear interpolation
- It is preferred to nearest point sampling not only because the final result suffers much less from the blocky image problem but also because the graphics hardware is usually optimized for bilinear interpolation.

\[ c_a = (1-p)c_1 + pc_2 \]
\[ c_b = (1-p)c_3 + pc_4 \]
\[ c = (1-q)c_a + qc_b \]
Filtering for Minification

- Consider a checker-board image texture.
- If all pixels are surrounded by dark-gray texels, the textured quad appears dark gray. If all pixels are surrounded by light-gray texels, the textured quad appears light gray.
**Mipmap**

- Why do we have the problem in minification? Simply put, it is because the texture is larger than the screen-space primitive and therefore many texels are not involved in filtering.
- A solution - Decrease the texture size such that all texels are involved. The best case is that the pixel and texel counts are the same.
- For decreasing the texture size, *down-sampling* is adopted.

- Given an original texture of $2^l \times 2^l$ resolution, a texture pyramid of $(l + 1)$ levels is constructed, where the original texture is located at level 0.
- The pyramid is called a *mipmap*.
- The level to filter is named *level of detail*, and is denoted by $\lambda$. 
Mipmapping (cont’d)

- Observe that a pixel covers a rectangular area on the screen. For simplicity, take the area as square such that the entire screen can be considered to be tiled by an array of square pixels.
- Then, a pixel's projection onto the texture space is not a ‘point’ but an ‘area’ centered at \((s', t')\). The projected area is called the footprint of the pixel.
Mipmapping (cont’d)

- In the example, the quad and level-1 texture have the same size. A pixel's footprint covers 2x2 texels in level 0, but covers a single texel in level 1.
- When a pixel footprint covers \( mxm \) texels of level-0 texture, \( \lambda \) is set to \( \log_2 m \).
Mipmapping (cont’d)

- In the example, the screen-space quad and level-2 texture have the same size. A pixel's footprint covers exactly a single texel in level-2 texture, which is then filtered.

- In this and previous examples, observe that all texels are involved in filtering.
Mipmapping (cont’d)

- In the following example, \( \lambda = \log_2 3 = 1.585.. \), and so we see levels 1 and 2, more formally \( \lfloor \lambda \rfloor \) and \( \lceil \lambda \rceil \).
Mipmapping (cont’d)

- Between $\lfloor \lambda \rfloor$ and $\lceil \lambda \rceil$, which one to choose?
  - We can take the nearest level: $i_\lambda = \lfloor \lambda + 0.5 \rfloor$
    - At the level, we can take the nearest point.
    - In contrast, we can adopt bilinear interpolation.
  - We can take both of them and linearly interpolate the filtering results.
    - At each level, we can take the nearest point.
    - In contrast, if bilinear interpolation is performed at each level, this is called *trilinear interpolation*.
Mipmap in GL

- GL provides a method for automatically generating a mipmap: `glGenerateMipmap`.
- `void glGenerateMipmap(GLenum target)`
  - `target` can be `GL_TEXTURE_2D`
Mipmapping in GL

- GL provides `void glGenerateMipmap(GLenum target)` where `target` can be `GL_TEXTURE_2D`.

- Using the footprint size, the GPU itself makes a choice between minification and magnification but allows the user to choose what kind of filtering to use in each case. For this purpose, `glTexParameteri` is called.

- `glTexParameteri(GLenum target, GLenum pname, GLint param)`
  - `target` is either `GL_TEXTURE_2D` or `GL_TEXTURE_CUBE_MAP`
  - `pname` can be the following:
    - `GL_TEXTURE_MAG_FILTER`
    - `GL_TEXTURE_MIN_FILTER`
    - `GL_TEXTURE_WRAP_S`
    - `GL_TEXTURE_WRAP_T`
  - `param`
    - If `pname` is `GL_TEXTURE_MAG_FILTER`, `param` is either `GL_NEAREST` or `GL_LINEAR` (for bilinear interpolation)
Texturing in OpenGL ES

- `glTexParameteri(GLenum target, GLenum pname, GLint param)`
  - param
    - If `pname` is `GL_TEXTURE_MIN_FILTER`, `param` can be:
      - `GL_NEAREST`
      - `GL_LINEAR`
      - `GL_NEAREST_MIPMAP_NEAREST`
      - `GL_NEAREST_MIPMAP_LINEAR`
      - `GL_LINEAR_MIPMAP_NEAREST`
      - `GL_LINEAR_MIPMAP_LINEAR`

When we do not use the mipmap, the level-0 texture is filtered either by nearest point sampling or by bilinear interpolation.
Mipmapping - Examples

- Consider a mipmap of stripe texture, and a long thin quad.
- When the quad is orthogonal to the view direction, the texturing result will be as follows.

![Diagram showing mipmaps and a quad](image-url)
Mipmapping – Examples (cont’d)

- Let us change level-0 texture only, just for the visualization purpose.
Mipmapping – Examples (cont’d)

- Assume that the quad is angled obliquely away from the viewer.
- When both `MAG_FILTER` and `MIN_FILTER` are `NEAREST`

- When both `MAG_FILTER` and `MIN_FILTER` are `LINEAR`
Mipmapping – Examples (cont’d)

- Let’s take **NEAREST_MIPMAP_NEAREST** for **MIN_FILTER** while taking **LINEAR** for **MAG_FILTER**.
**Mipmapping – Examples (cont’d)**

- Let’s change `MIN_FILTER` while taking `LINEAR` for `MAG_FILTER`.