Introduction

A texture is a rectangular, 2d image that is mapped onto a polygon to provide the illusion of fine-grained geometric or colour details that are not part of the underlying polygon model.

A texture T uses axes u and v to identify individual texture elements (or texels). T(u,v) maps to an RGB colour (r,g,b), that is, a texture is normally just a 2d colour bitmap.

To paste a texture onto a polygon, a mapping between the polygon's (x,y,z) positions and (u,v) locations in the texture must first be defined. This allows vertex positions on a polygon to be assigned corresponding (u,v) values. As the polygon is rendered, the (u,v) values are interpolated across the face of the polygon. For each pixel in the rendered polygon, the pixel's (u,v) is used to lookup a colour in texture T at T(u,v). The colour (r,g,b) is then applied to modify the polygon's colour appearance at that pixel (e.g., overwrite, alpha blend, modulate, etc.)
Mapping Modes

Texture \((u,v)\) axes range \([0,1]\) across the texture. If a \(u\) or \(v\) < 0 or > 1, different methods can be used to remap the value into a valid texture location.

One common method is clamping. \((u,v)\) values < 0 are set to 0, and \((u,v)\) values > 1 are set to 1.

One special effect supported by clamping is to map a small detail onto a polygon. For example, suppose a texture contained a bullet hole with a small amount of window around it. This texture could be mapped to a large polygon representing a large window. The \((x,y,z)\) to \((u,v)\) mapping is defined to produce \((u,v)\) values < 0 or > 1 everywhere except at the small location where the bullet hole is desired. Clamping guarantees the window texture everywhere except where the bullet hole occurs:

\[
T:
\]

\[
\begin{array}{c}
\bullet \\
\end{array}
\]
Another common method is wrapping. Any $(u,v) < 0$ or $> 1$ is wrapped to a valid value. For example, $u=1.5$ wraps to $u=0.5$. $u=-1.5$ also wraps to $u=0.5$.

Specifically:

$$(u',v') = (u - \lfloor u \rfloor, v - \lfloor v \rfloor)$$

where $\lfloor w \rfloor$ returns the largest integer less than or equal to $w$.

Wrapping allows a small texture to be wrapped repeatedly across a large polygon. For example, a small piece of brick texture could be wrapped to produce a long brick wall or a brick facade on a building. For this to work, the texture must be periodic, that is, the top and bottom edges of the texture must "match" so they will combine without producing a visible seam. Similarly, the left and right edges must also match.
Texture Filtering

A texture T is defined on a discrete, regular-grid lattice. However, pixels in a polygon will normally specify \((u,v)\) values that do not lie exactly at a grid location. There are a number of different ways to decide what RGB value to return when this happens.

The simplest method is to return the colour at the closest grid point, that is:

\[
(u',v') = \left\lfloor u + 0.5 \right\rfloor, \left\lfloor v + 0.5 \right\rfloor
\]

However, nearest-neighbour filtering can produce blocky-looking textures, particularly along polygon boundaries. A more sophisticated technique uses bilinear interpolation to combine the colours at the four corners of the cell that contains the target \((u,v)\) value:

Given texture colours \(c_0, c_1, c_2, c_3\) as shown around the target \((u,v)\), and distances \(s\) and \(t\) to the lower-left corner, the resulting colour at \((u,v)\) is:

\[
(1-s)(1-t)c_0 + (1-s)t c_1 + s t c_2 + s(1-t)c_3
\]
Mipmaps

Even with bilinear interpolation, aliasing problems can appear during texture mapping, particularly when the polygon moves far from the viewer and occupies only a few pixels. This can produce visually discontinuous results, since large jumps in \((u,v)\) values begin to occur when the (small) polygon is rendered. Rather than interpolating over possibly non-neighbouring texels, a pyramid of textures of decreasing size is constructed. Given an initial texture size of \(2^n \times 2^n\), \(n\) mipmaps of size \(2^i \times 2^i\), \(0 \leq i < n\) are built. Each mipmap is pre-rendered by averaging or blurring texels in the original texture map (e.g., via a Gaussian filter).

"Mip" comes from the Latin "multum in parvo", which means "many things into a small place."

The resulting pyramid is used to select texels that match closely in size to the pixel being texture mapped. The choice of which texel(s) to use from which mipmap is based on the number of texels \(d\) that cover the pixel. This value can be computed using equations for perspective projection to convert the screen-space size of a polygon to its size in \((u,v)\) space. \(d\) is then the determinant of the first derivative of the \(u\) and \(v\) values. If \(d < 1\) (i.e., if the
number of texels needed to cover the pixel is 1 or less), we use the original 2
texture T and apply nearest-neighbour or bilinear interpolation as before.

If $d > 1$, multiple texels cover a single pixel. There are numerous methods to
select the appropriate RGB colour. Some common methods include:
– select the mipmap nearest to $d$, then select the texel nearest to $(x,y)$
– select the mipmap nearest to $d$, then use bilinear interpolation on the four
texels that surround $(x,y)$
– select the two mipmaps that bound $d$, then linearly interpolate the texels on
each nearest to $(x,y)$
– select the two mipmaps that bound $d$, use bilinear interpolation on the four
texels that surround $(x,y)$ on each mipmap, then linearly interpolate the
resulting colours
Multitexturing

Texture mapping a polygon sets or modifies the colours of each of its pixels. There is no reason why multiple textures cannot be mapped onto a single polygon. Each new texture modifies whatever colours were set as a result of the previous texture mapping step.

Static multitexturing precomputes textures, and maps them into a scene as it is rendered. Dynamic multitexturing computes textures on-the-fly, and can therefore include properties that depend on the view position, or on dynamic objects and lights contained in the scene.

An example of static multitexturing is adding lighting effects from static lights. "Light maps" that encode illumination contributions can be precomputed, stored as a texture, and applied to appropriate objects in the scene as it is rendered.

An example of dynamic multitexturing is adding shadows for moving objects and/or lights in a scene. The shadow is normally dynamically computed for each triangle, then applied to the triangle as a texture. This allows the rendering algorithm to identify at render time which triangles are in shadow, and what type of shadow to apply.
Texture Mapping in OpenGL

OpenGL provides numerous texture mapping routines to allow you to define, load, store, and apply textures in a variety of different ways. To use texture mapping in OpenGL, you must perform the following steps:

1. Create a texture object and specify a texture for that object.
2. Indicate how the texture is to be applied to each pixel in a polygon.
3. Enable texture mapping.
4. Draw the scene, supplying both geometric and texture coordinates.

Below is a very simple example that creates a 2D texture from an external RGB image source, then maps the texture to a rectangular polygon.

```c
void texture_map_poly( unsigned char *data, int w, int h ) {
    // data: Pointer to RGB texture image, one byte per colour
    // w, h: Width, height of texture image, both powers of 2

    GLuint texture_ID;

    // Generate new texture (and set its ID), bind all 2D texture
    // calls to that texture
    glGenTextures( 1, &texture_ID );
    glBindTexture( GL_TEXTURE_2D, texture_ID );

    glTexImage2D( GL_TEXTURE_2D, 0, level
```
3,         // Number of colour components in
texture
w,         // Width of texture (must be power
of 2)
h,         // Height of texture (must be power
of 2)
0,         // Border width (0 or 1)
GL_RGB,    // Format of data (RGB triples)
GL_UNSIGNED_BYTE, // Type of data (1 byte/colour
component)
data       // Pointer to texture data
);

// Choose to replace any existing pixel colour with
colour
// from texture

glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_REPLACE );

// Use bilinear interpolation (GL_LINEAR) when pixel
maps to
// area greater (GL_TEXTURE_MIN_FILTER) or less than
// (GL_TEXTURE_MAG_FILTER) one texel; wrap the texture
// (GL_REPEAT) in both the u (GL_TEXTURE_WRAP_S) and the
v
// (GL_TEXTURE_WRAP_T) directions

glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR );

glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR );

glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_REPEAT );

glTexParameterf( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_REPEAT);

    // Enable 2D texture mapping, define which texture to map
    // from
    glEnable( GL_TEXTURE_2D );

    // Draw a simple, texture-mapped rectangle

    glBegin( GL_POLYGON );
    glTexCoord2f( 0, 0 ); // Lower-left corner is (u,v)=(0,0)
    glVertex2f( -1, -1 );
    glTexCoord2f( 1, 0 ); // Lower-right corner is (u,v)=(1,0)
    glVertex2f( 1, -1 );
    glTexCoord2f( 1, 1 ); // Upper-right corner is (u,v)=(1,1)
    glVertex2f( 1, 1 );
    glTexCoord2f( 0, 1 ); // Upper-left corner is (u,v)=(0,1)
    glVertex2f( -1, 1 );
    glEnd();

    // Disable 2D texture mapping, free texture data
    
    disable( GL_TEXTURE_2D );
    deleteTextures( 1, &texture_ID );

    }

OpenGL supports numerous additional options. For example, if you want to generate and use mipmaps, this can be done automatically by replacing the glTexImage2D call with:

    gluBuild2DMipmaps( GL_TEXTURE_2D, 3, w, h, GL_RGB,
GL_UNSIGNED_BYTE, data);

If you use mipmaps, you need to change the way you minify and magnify when a pixel maps to more than or less than a single texel:

    glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_NEAREST );
        glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR );

These commands tell OpenGL to use bilinear interpolation when the pixel covers less than one texel (GL_LINEAR for magnify), and to select the nearest mipmap and perform bilinear interpolation within that mipmap when the pixel covers more than one texel (GL_LINEAR_MIPMAP_NEAREST for minify).

OpenGL has many additional options to control the creation and application of 1D, 2D, and 3D textures. See Chapter 9 of the OpenGL Programming Guide (pp. 359–450) for complete details.