1 Texture

The most common form of texture mapping means laying a color pattern defined by an image onto a 3D surface. It allows to render 3D surfaces with realistic color variation. From programmer's perspective, this is done by providing texture coordinates for each vertex. For each triangle of the 3D object the texture applied to it is copied from the triangle bounded by points given by the texture coordinates (see Figure 1). More precisely, texture coorinates are linearly interpolated when the triangle is scan-converted. Then, on the pixel processing stage, they are used to look up color from the texture image. Texture coordinates $(0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$ correspond to the corners of the image. There are several ways to handle the case of one or both coordinates being outside of the range from 0 to 1. For example, one can use only the fractional part of each coordinate to do texture lookup or they can both be clamped to $[0, 1]$ (see 'Repeating and Clamping Textures' in Chapter 9 of the OpenGL programming guide).

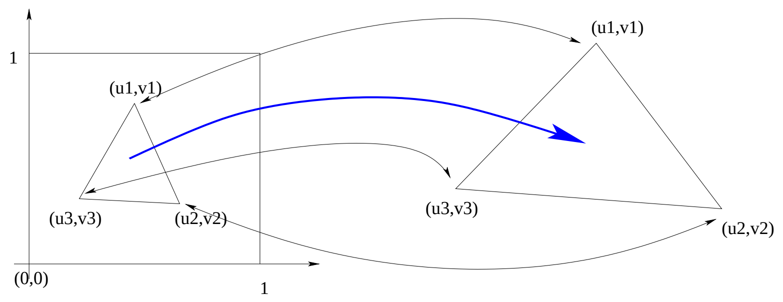

Figure 1: Applying a texture (left) to a 3D triangle (right). (u_i, v_i) are the texture coordinates of the vertices. The contents (colors) of the image specified as the texture are copied to the 3D triangle. Note that this requires stretching or shrinking the texture, particularly if the two triangles differ a lot in shape or size.

OpenGL allows to use up to 4 texture coordinates and they can be transformed using a 4×4 texture matrix. Eventually (i.e. after this transformation), only two texture coordinates are used.

Apart from 2D trextures, one can also use one-dimensional textures or three dimensional textures. In one-dimensional case, only one texture coordinate is used to look up the color and the texture is one-dimensional. In the three dimensional case, the texture is 3D (think of it as a colored volume) and three texture coordinates are used for color lookup. The simplest application of 3D texture is to use texture that describes color distribution in some material, e.g. wood, concrete or marble and use vertex coordinates as texture coordinates. This causes a color of a point on the 3D surface to be looked up from its location in the volume and produces an object looking as if it was carved from the volume.

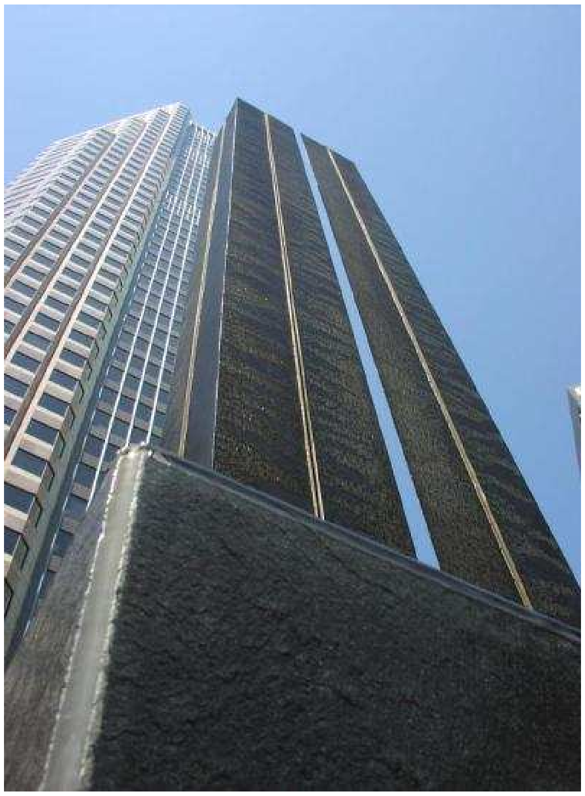

Figure 2: Sizes of windows depend on how far they are.

2 Perspectively correct interpolation

In graphics pipeline setting, texture coodinates enter the pipeline as vertex attributes. They are interpolated in the rasterization stage. Interpolated texture coordinates are passed on as fragment attributes. In the fragment processing stage, the color of each fragment is looked up from the texture image (the 1D or 3D cases work the same way, except for 1D or 3D texture is used instead of image/2D texture).

2.1 Inadequacy of screen space interpolation of texture coordinates

So far (as a way to compute depth or color of fragments) we were using screen space linear interpolation. The data to be interpolated was sent with vertices in screen coordinates from vertex processor to the rasterizer. The rasterizer interpolated this data from projected vertices to fragments. This happened for depth, color in Gouraud shading and normals in Phong shading.

However, this is not the right approach for textures (in fact, it is not completely correct for shading either, but its incorrectness is less aparent in this case). Imagine a tall building with uniformly regular distribution and equal sizes of windows (all 'normal' tall buildings are like this). Walk close to it and look up. The windows are going to appear smaller on higher floors (Figure 2). Let's say you want to render this building as a big rectangular paralleliped using a texture to model color variation on the facade (including the windows). On this texture, all windows are going to be of the same size (since they are of the same size in 3D). If you were to use screen-space linear interpolation to interpolate texture coordinates, they would also be of the same size on the image.

To sum up, what we need to do for texture coordinates is to first linearly interpolate them in 3D and then project the result to 2D (this process is called *perspectively correct* interpolation). If we do things the usual way: first project vertices and then linearly interpolate texture coordinates from projected vertices, we'll get wrong results.

2.2 Perspectively correct interpolation

We'll argue here that perspectively correct interpolation is equivalent to linear interpolations and division. Recall that we would like to achieve the result equivalent to linear interpolation (of texture coordinates) in 3D and then project the results to 2D, not the other way around (which would be equivalent to linear interpolation in screen space).

Let's go back to reduction of a general perspective view to the canonical view. Break it into two stages: everything that happens before the nonlinear part (arrow with the perspective transformation matrix next to it on the figure in transformation notes), and the nonlinear part together with everything that happens after it.

The first part is purely affine. Therefore it can be described using a 4×4 transformation matrix A in homogenous coordinates. If the coordinates of the result are denoted by x_A, y_A, z_A, we can write the transformation as follows

$$\begin{bmatrix} x_A \\ y_A \\ z_A \\ 1 \end{bmatrix} = A * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}.$$

The second part of the transformation maps the result of the first one to screen coordinates. To simplify things, we'll write it using the proportionality relation of vectors (or points), which we denote by $\approx$. Thus, $v \approx w$ means that one can scale v using a nonzero constant to get w ($w = cv$ for some $c \neq 0$). The transformation we are looking at maps (x_A, y_A, z_A) into $(x_A/z_A, y_A/z_A, l(k - 1/z_A))$ (since it is the simple form of perspective transformation followed by scaling and translation in z), where k and l are some constants. This point is proportional to $(x_A, y_A, (kl)z_A - l)$. So, if we denote the coordinates of the resulting point by (x', y', z'), we can write

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \approx P * \begin{bmatrix} x_A \\ y_A \\ z_A \\ 1 \end{bmatrix},$$

where P is in the following simple 3×4 matrix:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & kl & -l \end{bmatrix}.$$

Putting all of the above together, we can write the entire transformation as

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \approx M * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{1}$$

where $M = P * A$ is a 3×4 matrix.

Now, let's say we have a triangle with texture coordinates in 3D. We want to linearly interpolate texture coordinates to points on this triangle. This process can be described as an assignment of a 3D point to texture coordinates (u, v). The 3D point corresponding to (u, v) should get color the texture image has at (u, v). Since we use linear interpolation, this defines an affine mapping of texture coordinates into 3D (in Figure 1, this mapping is shown as the blue arrow), which can be thought of as a parametrization of the triangle over texture coordinates. This mapping can be represented using a 4×3 matrix T:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = T * \begin{bmatrix} u \\ w \\ 1 \end{bmatrix}.$$

Using this equation with (1) we obtain one that essentially describes how the texture image needs to be mapped onto the screen:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \approx M * T * \begin{bmatrix} u \\ w \\ 1 \end{bmatrix}.$$

The matrix M depends only on the view, so it can be computed when the view is specified (just once before rendering all triangles). The matrix T has to be computed on per-triangle basis, based on the coordinates and texture coordinates of its vertices. Notice that $B = M * T$ is a 3×3 matrix. If $B = [b_{ij}]_{i=1\ldots3, j=1\ldots3}$, we can write

$$\begin{bmatrix} u \\ w \\ 1 \end{bmatrix} \approx \begin{bmatrix} b_{22}b_{33} - b_{23}b_{32} & b_{13}b_{32} - b_{12}b_{33} & b_{12}b_{23} - b_{13}b_{22} \\ b_{23}b_{31} - b_{21}b_{33} & b_{11}b_{33} - b_{13}b_{31} & b_{21}b_{13} - b_{11}b_{23} \\ b_{21}b_{32} - b_{22}b_{31} & b_{12}b_{31} - b_{11}b_{32} & b_{11}b_{22} - b_{21}b_{12} \end{bmatrix} * \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}. \tag{2}$$

The 3×3 matrix in the above equation plays the role of the inverse of B: even though it is not the same as the inverse of B, it is proportional to it (which is enough for the equation hold up to a constant factor; note it makes things possible to carry out also in some degenerate cases in which B is not invertible and is also faster since it does not require division or determinant calculation).

In fact, this equation (2) also describes precisely what the rasterizer needs to do to determine the perspectively correct texture coordinates that need to be used with each fragment. Let's say the rasterizer generates a fragment at (p_x, p_y). It knows how to compute its depth p_z. Now, it has all three coordinates of the fragment in screen coordinates. It evaluates the right-hand side of (2) to compute a vector proportional to $\begin{bmatrix} u \\ w \\ 1 \end{bmatrix}$. Let's say this vector has coordinates $[u_*, v_*, w_*]$. Then, u can be computed as u_*/w_* and v - as v_*/w_*.

An important thing to note here is that u_*, v_*, w_* are given by linear expressions in x' and y' (equation (2) shows they are linear in x', y', z' and we know that z', i.e. depth, is linear in x' and y'). Thus, to perform perspectively correct interpolation of texture coordinates, the rasterizer needs to do *three linear interpolations* (needeed to compute u_*, v_*, w_*) and do *two divisions* for each fragment. The coefficients needed for linear interpolation can be precomputed on per-triangle basis for efficient computation in scanline order.