texture coordinates in screen space results in incorrect images, as shown for the grid texture shown in Figure 11.11. Because things in perspective get smaller as the distance to the viewer increases, the lines that are evenly spaced in 3D should compress in 2D image space. More careful interpolation of texture coordinates is needed to accomplish this.

11.3.1 Perspective Correct Textures

We can implement texture mapping on triangles by interpolating the (u, v) coordinates, modifying the rasterization method of Section 8.1.2, but doing this without accounting for perspective results in the problem shown at the right of Figure 11.11. A similar problem occurs for triangles if screen-space barycentric coordinates are used as in the following rasterization code:

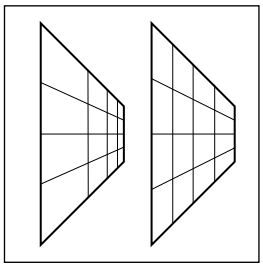

Figure 11.11. Left: correct perspective. Right: interpolation in screen space.

```
for all x_s do
  for all y_s do
    compute (α, β, γ) for (x_s, y_s)
    if α ∈ (0,1) and β ∈ (0,1) and γ ∈ (0,1) then
      t = αt_0 + βt_1 + γt_2
      drawpixel (x_s, y_s) with color texture(t) for a solid texture
      or with texture(β, γ) for a 2D texture.
```

This code will generate images, but there is a problem. To unravel the basic problem, let's consider the progression from a world space point $\mathbf{q}$ to the homogeneous screen-space point $\mathbf{r}$ to the homogenized screen-space point $\mathbf{s}$:

$$\begin{bmatrix} x_q \\ y_q \\ z_q \\ 1 \end{bmatrix} \xrightarrow{\text{transform}} \begin{bmatrix} x_r \\ y_r \\ z_r \\ w_r \end{bmatrix} \xrightarrow{\text{homogenize}} \begin{bmatrix} x_r/w_r \\ y_r/w_r \\ z_r/w_r \\ 1 \end{bmatrix} \equiv \begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix}.$$

The simplest form of the texture coordinate interpolation problem is when we have texture coordinates (u, v) associated with two points, $\mathbf{q}$ and $\mathbf{Q}$, and we need to generate texture coordinates in the image along the line between $\mathbf{s}$ and $\mathbf{S}$. If the world-space point $\mathbf{q}'$ that is on the line between $\mathbf{q}$ and $\mathbf{Q}$ projects to the screen-space point $\mathbf{s}'$ on the line between $\mathbf{s}$ and $\mathbf{S}$, then the two points should have the same texture coordinates.

The naïve screen-space approach, embodied by the algorithm above, says that at the point $\mathbf{s}' = \mathbf{s} + \alpha(\mathbf{S} - \mathbf{s})$ we should use texture coordinates $u_s + \alpha(u_S - u_s)$ and $v_s + \alpha(v_S - v_s)$. This doesn't work correctly because the world-space point $\mathbf{q}'$ that transforms to $\mathbf{s}'$ is *not* $\mathbf{q} + \alpha(\mathbf{Q} - \mathbf{q})$.

However, we know from Section 7.4 that the points on the line segment between $\mathbf{q}$ and $\mathbf{Q}$ do end up somewhere on the line segment between $\mathbf{s}$ and $\mathbf{S}$; in fact, in that section we showed that

$$\mathbf{q} + t(\mathbf{Q} - \mathbf{q}) \mapsto \mathbf{s} + \alpha(\mathbf{S} + \mathbf{s}).$$

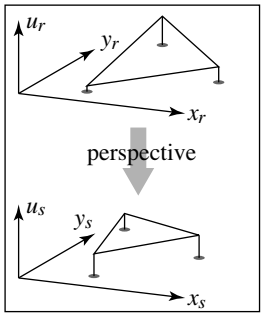

Figure 11.12. Geometric reasoning for screen-space interpolation. Top: u_r is to be interpolated as a linear function of (x_r, y_r). Bottom: after a perspective transformation from (x_r, y_r, u_r, w_r) to $(x_s, y_s, u_s, 1)$, u_s is a linear function of (x_s, y_s).

The interpolation parameters t and α are not the same, but we can compute one from the other:[1]

$$t(\alpha) = \frac{w_r \alpha}{w_R + \alpha(w_r - w_R)} \quad \text{and} \quad \alpha(t) = \frac{w_R t}{w_r + t(w_R - w_r)}. \tag{11.4}$$

These equations provide one possible fix to the screen-space interpolation idea. To get texture coordinates for the screen-space point $\mathbf{s}' = \mathbf{s} + \alpha(\mathbf{S} - \mathbf{s})$, compute $u'_s = u_s + t(\alpha)(u_S - u_s)$ and $v'_s = v_s + t(\alpha)(v_S - v_s)$. These are the coordinates of the point $\mathbf{q}'$ that maps to $\mathbf{s}'$, so this will work. However, it is slow to evaluate $t(\alpha)$ for each fragment, and there is a simpler way.

The key observation is that because, as we know, the perspective transform preserves lines and planes, it is safe to linearly interpolate any attributes we want across triangles, but only as long as they go through the perspective transformation along with the points. To get a geometric intuition for this, reduce the dimension so that we have homogeneous points (x_r, y_r, w_r) and a single attribute u being interpolated. The attribute u is supposed to be a linear function of x_r and y_r, so if we plot u as a height field over (x_r, y_r) the result is a plane. Now, if we think of u as a third spatial coordinate (call it u_r to emphasize that it's treated the same as the others) and send the whole 3D homogeneous point (x_r, y_r, u_r, w_r) through the perspective transformation, the result (x_s, y_s, u_s) still generates points that lie on a plane. There will be some warping within the plane, but the plane stays flat. This means that u_s is a linear function of (x_s, y_s)—which is to say, we can compute u_s anywhere by using linear interpolation based on the coordinates (x_s, y_s).

Returning to the full problem, we need to interpolate texture coordinates (u, v) that are linear functions of the world space coordinates (x_q, y_q, z_q). After transforming the points to screen space, and adding the texture coordinates as if they were additional coordinates, we have

$$\begin{bmatrix} u \\ v \\ 1 \\ x_r \\ y_r \\ z_r \\ w_r \end{bmatrix} \xrightarrow{\text{homogenize}} \begin{bmatrix} u/w_r \\ v/w_r \\ 1/w_r \\ x_r/w_r = x_s \\ y_r/w_r = y_s \\ z_r/w_r = z_s \\ 1 \end{bmatrix} \tag{11.5}$$

[1] It is worth while to derive these functions yourself from Equation (7.6); in that chapter's notation, $\alpha = f(t)$.

The practical implication of the previous paragraph is that we *can* go ahead and interpolate all of these quantities based on the values of (x_s, y_s)—including the value z_s, used in the z-buffer. The problem with the naïve approach is simply that we are interpolating components selected inconsistently—as long as the quantities involved are from before or all from after the perspective divide, all will be well.

The one remaining problem is that $(u/w_r, v/w_r)$ is not directly useful for looking up texture data; we need (u, v). This explains the purpose of the extra parameter we slipped into (11.5), whose value is always 1: once we have u/w_r, v/w_r, and $1/w_r$, we can easily recover (u, v) by dividing.

To verify that this is all correct, let's check that interpolating the quantity $1/w_r$ in screen space indeed produces the reciprocal of the interpolated w_r in world space. To see this is true, confirm (Exercise 2):

$$\frac{1}{w_r} + \alpha(t)\left(\frac{1}{w_R} - \frac{1}{w_r}\right) = \frac{1}{w'_r} = \frac{1}{w_r + t(w_R - w_r)} \tag{11.6}$$

remembering that $\alpha(t)$ and t are related by Equation 11.4.

This ability to interpolate $1/w_r$ linearly with no error in the transformed space allows us to correctly texture triangles. We can use these facts to modify our scan-conversion code for three points $\mathbf{t}_i = (x_i, y_i, z_i, w_i)$ that have been passed through the viewing matrices, but have not been homogenized, complete with texture coordinates $\mathbf{t}_i = (u_i, v_i)$:

for all x_s **do**
  for all y_s **do**
    compute (α, β, γ) for (x_s, y_s)
    if ($\alpha \in [0, 1]$ and $\beta \in [0, 1]$ and $\gamma \in [0, 1]$) **then**
      $u_s = \alpha(u_0/w_0) + \beta(u_1/w_1) + \gamma(u_2/w_2)$
      $v_s = \alpha(v_0/w_0) + \beta(v_1/w_1) + \gamma(v_2/w_2)$
      $1_s = \alpha(1/w_0) + \beta(1/w_1) + \gamma(2/w_2)$
      $u = u_s/1_s$
      $v = v_s/1_s$
      drawpixel (x_s, y_s) with color texture(u, v)

Of course, many of the expressions appearing in this pseudocode would be precomputed outside the loop for speed. For solid textures, it's simple enough to include the original world space coordinates x_q, y_q, z_q in the list of attributes, treated the same as u and v, and correct interpolated world space coordinates will be obtained, which can be passed to the solid texture function.