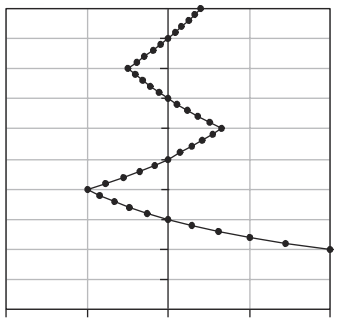

CHAPTER TEN

W Pleasure, W Fun

MAY–JUNE 1998

The whole point of doing 3D graphics is the third dimension. But since the screen is only 2D, the third dimension appears only indirectly in terms of perspective and occlusion. Correct occlusion testing is, however, fairly sensitive to precision problems in the depth calculation. In this chapter, I will review the traditional way to represent depth and introduce a new technique that appears in the new generation of 3D graphics boards. This technique has become practical as a side effect of perspective-correct texture-mapping hardware. Both ways have their good and bad points, so I'll finish up by establishing some rules of thumb on which to choose in a given situation.

Mathematical Niceties

To simplify things a bit in this discussion, I'm not going to include the *y* coordinates in any calculations. The problem can be adequately understood in terms of only the *x*, *z*, and *w* coordinates, and the reduction in dimensionality will simplify things considerably.

Next, let's define our coordinate systems. There are three of interest to us:

1. *Eye space*: All objects are translated so that the eye is at the origin and is looking down the positive *z* axis (this, incidentally, is a left-handed coordinate system).
2. *Perspective space*: This occurs after multiplying points in eye space by a homogeneous perspective transformation.

3. *Screen space*: This occurs after dividing out the w component of the perspective space points.

Finally, there is the question of notation. A mathematical symbol can convey a lot of information if you give it a chance. The mathematical symbols I use here will designate coordinates of various points in various coordinate systems. The three things, then, that we want to explicitly convey are

- The name of the point
- The component (x, z, or w)
- The coordinate system

The symbology available to us consists of letters, subscripts, and other mathematical decorations applied to letters. I will use the following choices:

- The component will be designated by the main letter variable: x, z, or w.
- The coordinate system will be a decoration over the letter as follows:
  x (a bare letter) means eye space
  $\hat{x}$ means perspective space before w division
  $\tilde{x}$ means screen space (perspective space after w division)
  Essentially, the number of wiggles over a letter tell how many transformations it has gone through.
- The name of the point will be a subscript.

For example, the z coordinate of point 0 in perspective space will be denoted as $\hat{z}_0$. Any equations with coordinates that appear without subscripts will indicate generic relations that apply to all points.

Traditional Perspective

I described the derivation of the homogeneous perspective matrix in Chapters 3 and 18 of *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*. If we only consider the x, z, and w components, a typical homogeneous perspective matrix looks like

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & d \\ 0 & c & 0 \end{bmatrix}$$

This gives the following generic relations between components in the various coordinate systems.

Eye to perspective:

$$\begin{bmatrix} x & z & 1 \end{bmatrix} \begin{bmatrix} a & 0 & 0 \\ 0 & b & d \\ 0 & c & 0 \end{bmatrix} = \begin{bmatrix} ax & bz + c & dz \end{bmatrix} = \begin{bmatrix} \hat{x} & \hat{z} & \hat{w} \end{bmatrix}$$

Perspective to screen:

$$\begin{bmatrix} \frac{\hat{x}}{\hat{w}} & \frac{\hat{z}}{\hat{w}} \end{bmatrix} = \begin{bmatrix} \tilde{x} & \tilde{z} \end{bmatrix}$$

or, composing these two:

Eye to screen:

$$\begin{bmatrix} \frac{ax}{dz} & \frac{bz + c}{dz} \end{bmatrix} = \begin{bmatrix} \tilde{x} & \tilde{z} \end{bmatrix}$$

A key property of the homogeneous perspective transform is revealed when we examine what happens to a straight line segment in eye space when it is transformed into screen space. The line segment generated by linear interpolation between eye space endpoints $\begin{bmatrix} x_0 & z_0 \end{bmatrix}$ and $\begin{bmatrix} x_1 & z_1 \end{bmatrix}$ can be represented by the parametric equation

$$\begin{bmatrix} x_L & z_L \end{bmatrix} = \begin{bmatrix} x_0 + \alpha(x_1 - x_0) & z_0 + \alpha(z_1 - z_0) \end{bmatrix}$$

What does this shape transform into in screen space? Plug in the eye-to-screen transform equation and we get

$$\begin{bmatrix} \tilde{x}_L & \tilde{z}_L \end{bmatrix} = \begin{bmatrix} \frac{a(x_0 + \alpha(x_1 - x_0))}{d(z_0 + \alpha(z_1 - z_0))} & \frac{b(z_0 + \alpha(z_1 - z_0)) + c}{d(z_0 + \alpha(z_1 - z_0))} \end{bmatrix}$$

At first, this might look fairly mysterious. Our shape is a parametric curve with the x and z coordinates generated by hyperbolic functions of α. It never ceases to amaze me that plotting one hyperbola against the other yields . . . a straight line. Over the years, I have collected various ways to show this. The simplest is just to solve $\tilde{x}_L$ for alpha and plug that into the expression for $\tilde{z}_L$ to get a linear equation in $\tilde{x}_L$ and $\tilde{z}_L$. You can never have too many visualizations, however, so here is another one more appropriate for the current discussion. The exposition will be easier to manage if we write the expressions for $\tilde{x}_L$ and $\tilde{z}_L$ as

$$\tilde{x}_L = \frac{A + B\alpha}{E + F\alpha} \qquad \tilde{z}_L = \frac{C + D\alpha}{E + F\alpha}$$

The key thing to note is that both $\tilde{x}_L$ and $\tilde{z}_L$ have the same denominator. The reason this is significant is that it places the asymptotes of $\tilde{x}_L$ and $\tilde{z}_L$ at the same place: $\alpha = -\text{F/E}$. We can therefore move both asymptotes to the origin by changing the parameterization via the replacement:

$$\alpha' = E + F\alpha$$

Putting this into the equation gives

$$\tilde{x}_L = \frac{A + B\left(\frac{\alpha' - E}{F}\right)}{\alpha'} = \left(\frac{B}{F}\right) + \frac{1}{\alpha'}\left(\frac{AF - BE}{F}\right)$$

$$\tilde{z}_L = \frac{C + D\left(\frac{\alpha' - E}{F}\right)}{\alpha'} = \left(\frac{D}{F}\right) + \frac{1}{\alpha'}\left(\frac{CF - DE}{F}\right)$$

Now this is much more obviously the parametric equation of a straight line segment. And it brings up another important property: equally spaced points in eye space (equal steps in α, and therefore in α') transform into nonequally spaced points in screen space (as evinced by the $1/\alpha'$ term). We'll make more of this later.

Anyway, we can now see that straight lines transform into straight lines and, more generally, flat polygons transform into flat polygons. This means that we can calculate values of $\tilde{z}$ at just the triangle vertices and linearly interpolate these values in screen space in order to find the proper value for depth comparisons. A depth buffer implementation of this would properly be called a $\tilde{z}$ buffer in our notation. Figures 10.1(a) and 10.1(b)

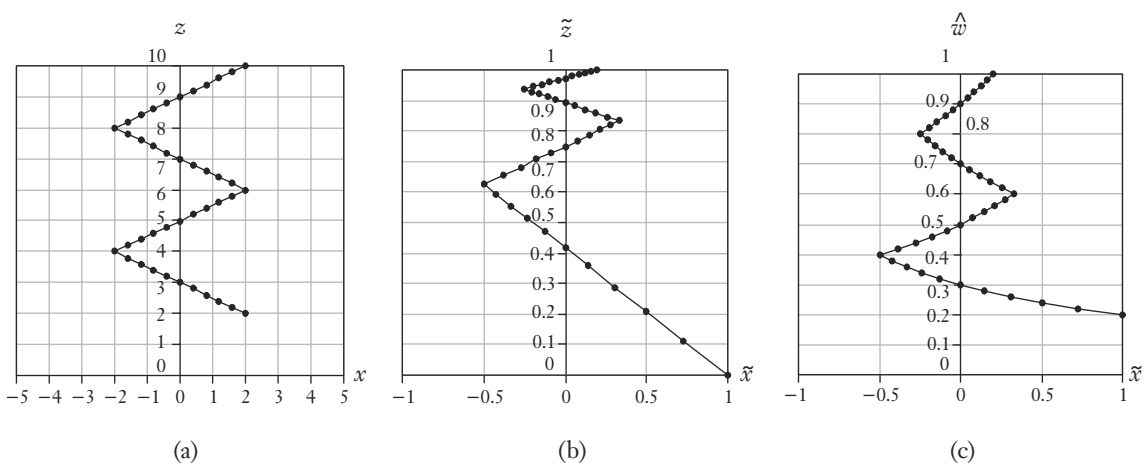

Figure 10.1 *Transformation of straight lines in eye space: eye space (a), screen space (b), and $\hat{w}$ space (c)*

show the distortion induced on a couple of straight segments due to this perspective transform (ignore Figure 10.1(c) for now).

$\tilde{z}$ resolution

Now it's time to bow to reality and take a look at some resolution issues. If we use the value of $\tilde{z}$ for depth comparisons, how does resolution in $\tilde{z}$ translate into resolution in eye space z? Start with the mapping from eye space z to screen space $\tilde{z}$. We have, generically

$$\tilde{z} = \frac{bz + c}{dz} = \left(\frac{b}{d}\right) + \frac{1}{z}\left(\frac{c}{d}\right)$$

In other words, the $\tilde{z}$ coordinate is a scale and offset applied to one over the eye space z coordinate. The values of the scale and offset will determine the range of values we expect to have to deal with for $\tilde{z}$. It is typical to specify these scales and offsets in terms of two depth values in eye space called z-near, z_n, and z-far, z_f, that will map to $\tilde{z}_n = 0$ and $\tilde{z}_f = 1$. A little brain exercise will show that, in order to make this happen, we must have

$$b = d\left(\frac{z_f}{z_f - z_n}\right)$$
$$c = d\left(\frac{-z_n z_f}{z_f - z_n}\right)$$

Putting these together gives the following, which I've written in a whole buncha different ways:

$$\begin{aligned}
\tilde{z} &= \frac{bz + c}{dz} \\
&= \left(\frac{z_f}{z_f - z_n}\right) - \frac{1}{z}\left(\frac{z_n z_f}{z_f - z_n}\right) \\
&= \left(\frac{z_f}{z_f - z_n}\right)\frac{z - z_n}{z} \\
&= \frac{\frac{1}{z_n} - \frac{1}{z}}{\frac{1}{z_n} - \frac{1}{z_f}}
\end{aligned}$$

In a typical application, we would pick z_n and z_f to bracket our data in eye space z, set up the matrix appropriately, and throw a bunch of points down it, knowing we will get $\tilde{z}$ values between 0 and 1.

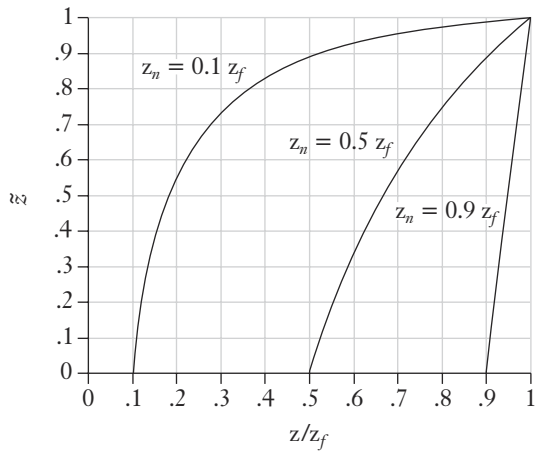

Figure 10.2 *Distortion of $\tilde{z}$ for various values of z_n*

Figure 10.2 shows the mapping from eye space z to screen space $\tilde{z}$ for various values of z_n as a proportion of z_f. We note that if z_n is much smaller than z_f, the values of $\tilde{z}$ are all smushed together near the value 1, a prospect that is dangerous to the health of our resolution. The recommended practice is to make z_n as far from the eye as possible. This gives the more linear relationship of the curve at the right side of Figure 10.2. Now let's see explicitly how the choice of z_n affects depth resolution.

Suppose we use a fixed-point representation for $\tilde{z}$. The values we can represent are equally spaced along the $\tilde{z}$ axis. That's all very nice, but it's physically more meaningful to see how these quantization bins look in eye space. We can find these by mapping equal steps in $\tilde{z}$ back to the z axis. The different choices for z_n from Figure 10.2 will give the different quantization spacings shown in Figure 10.3(a) using (for clarity) 16 quantization steps. You can see that a low value for z_n gives a rather bad situation. Detail on objects that are far from the eye can all map into the same quantized $\tilde{z}$ value.

If we happen to use floating point for $\tilde{z}$, the situation is even worse. Here, our 16 quantization steps are more or less logarithmically spaced along $\tilde{z}$ in a way that spreads things out even more at large distances. Figure 10.3(b) shows the logarithmic spacing of floating-point values exacerbating the nonlinear spacing of z values.

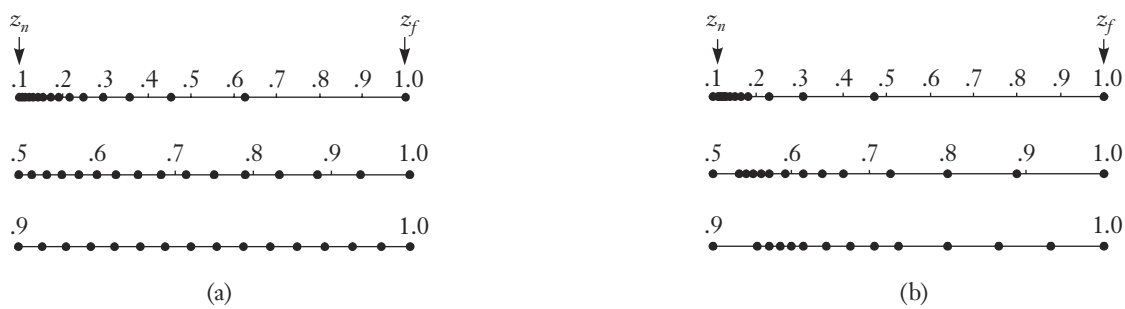

Figure 10.3 *Quantization of z space: fixed point $\tilde{z}$ (a) and floating point $\tilde{z}$ (b)*

Texture Mapping

Now, off on another tangent for a bit.

If you are performing texture mapping on a triangle, it is necessary to calculate, for each pixel, the appropriate u, v texture coordinate values. You generally specify the u and v at triangle vertices with the implicit assumption that they will be linearly interpolated across the triangle in eye space. That is, u and v are linear functions of eye space x and z. But when rendering, we want to calculate u and v as functions of the screen space $\tilde{x}$. How do we do this? I gave one derivation in Chapter 17, "Hyperbolic Interpolation" of *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*. Here's a different one.

What we have is two categories of parameter. The first category is x and z and any others that are linear functions of x and z. These include texture coordinates u and v, and perspective space coordinates $\hat{x}$, $\hat{z}$, and $\hat{w}$. If we linearly interpolate x_L and z_L by the parameter α according to

$$\begin{aligned} x_L &= x_0 + \alpha(x_1 - x_0) \\ z_L &= z_0 + \alpha(z_1 - z_0) \end{aligned}$$

then we can find all these other parameters by interpolating with the same α value:

$$\begin{aligned} \hat{x}_L &= \hat{x}_0 + \alpha(\hat{x}_1 - \hat{x}_0) \\ \hat{w}_L &= \hat{w}_0 + \alpha(\hat{w}_1 - \hat{w}_0) \\ u_L &= u_0 + \alpha(u_1 - u_0) \end{aligned}$$

The second category of parameters consists of the screen space coordinates $\tilde{x}$ and $\tilde{z}$, which are *not* simple linear combinations of x and z. When stepping across a scan line, we interpolate between screen space endpoints by using uniform steps of β in the formulas.

$$\begin{aligned} \tilde{x}_L &= \tilde{x}_0 + \beta(\tilde{x}_1 - \tilde{x}_0) \\ \tilde{z}_L &= \tilde{z}_0 + \beta(\tilde{z}_1 - \tilde{z}_0) \end{aligned} \tag{10.1}$$

We desire a relation between α and β. Going back to the definition of $\tilde{z}_L$, we can write it in terms of α as

$$\tilde{z}_L = \frac{\hat{z}_L}{\hat{w}_L} = \frac{\hat{z}_0 + \alpha(\hat{z}_1 - \hat{z}_0)}{\hat{w}_0 + \alpha(\hat{w}_1 - \hat{w}_0)}$$

Now pull in the transformation equations from perspective to screen space, modified slightly:

$$\hat{z}_0 = \tilde{z}_0\,\hat{w}_0$$
$$\hat{z}_1 = \tilde{z}_1\,\hat{w}_1$$

so that

$$\tilde{z}_L = \frac{\tilde{z}_0\,\hat{w}_0 + \alpha\left(\tilde{z}_1\,\hat{w}_1 - \tilde{z}_0\,\hat{w}_0\right)}{\hat{w}_0 + \alpha\left(\hat{w}_1 - \hat{w}_0\right)}$$

Add and subtract $\alpha\tilde{z}_0\,\hat{w}_1$ to the numerator and fiddle around a while, and you can convert this to

$$\tilde{z}_L = \tilde{z}_0 + \left(\frac{\alpha\,\hat{w}_1}{\hat{w}_0 + \alpha\left(\hat{w}_1 - \hat{w}_0\right)}\right)\left(\tilde{z}_1 - \tilde{z}_0\right)$$

Comparing this with Equation (10.1) gives us the following relation connecting interpolation in eye space with interpolation in screen space:

$$\beta = \frac{\alpha\,\hat{w}_1}{\hat{w}_0 + \alpha\left(\hat{w}_1 - \hat{w}_0\right)}$$

Or, solving for α:

$$\alpha = \frac{\beta\,\hat{w}_0}{\hat{w}_1 + \beta\left(\hat{w}_0 - \hat{w}_1\right)}$$

Now we can calculate u_L in terms of β. Plug and shuffle to get

$$\begin{aligned} u_L &= u_0 + \alpha\left(u_1 - u_0\right) \\ &= u_0 + \frac{\beta\hat{w}_0}{\hat{w}_1 + \beta\left(\hat{w}_0 - \hat{w}_1\right)}\left(u_1 - u_0\right) \\ &= \frac{u_0\hat{w}_1 + \beta\left(u_1\hat{w}_0 - u_0\hat{w}_1\right)}{\hat{w}_1 + \beta\left(\hat{w}_0 - \hat{w}_1\right)} \\ &= \frac{\dfrac{u_0}{\hat{w}_0} + \beta\left(\dfrac{u_1}{\hat{w}_1} - \dfrac{u_0}{\hat{w}_0}\right)}{\dfrac{1}{\hat{w}_0} + \beta\left(\dfrac{1}{\hat{w}_1} - \dfrac{1}{\hat{w}_0}\right)} \end{aligned}$$

Now it's time to take a deep breath and interpret this result. We calculate the values of $\left(u/\hat{w}\right)$ and $\left(1/\hat{w}\right)$ at each endpoint of a line (or at each vertex of a triangle) and then linearly interpolate *these* values in screen space. Divide them to get u_L. This mechanism works for any parameter

that is linear in eye space. Since we are going to have to do this for several such values (u, v, and perhaps even vertex colors), it is best to calculate the value of $\hat{w}$ once and then multiply it by the interpolated values of $(u/\hat{w})$. The interpolation equation for $\hat{w}$ is

$$\hat{w}_L = \frac{1}{\frac{1}{\hat{w}_0} + \beta\left(\frac{1}{\hat{w}_1} - \frac{1}{\hat{w}_0}\right)}$$

so we linearly interpolate the denominator, that is, $(1/\hat{w}_L)$, and do just one division (per pixel) to get $\hat{w}_L$.

W Buffering

The calculations necessary for texture mapping give us a new way to do depth testing. Since we are going to calculate $\hat{w}$ anyway, doing a divide per pixel, we can simply use $\hat{w}$ (which is really a scaled version of eye space z) for depth testing. That is, instead of doing $\tilde{z}$ buffering, we would be doing $\hat{w}$ buffering. Note that this is not the same as the classic error of simply interpolating $\hat{w}$ (or z) linearly in screen space. $\hat{w}$ buffering works properly only if we calculate $\hat{w}$ as the correct hyperbolic function of β (and hence of screen x). This means that flat lines and planes are no longer flat in a $\hat{w}$ buffer scheme. Figure 10.1(c) shows the $\hat{w}$ buffer version of Figures 10.1(a) and 10.1(b).

Note that with $\hat{w}$ buffering, we do not need to specify a value for z_n. That's good because this parameter has always been confusing to computer graphics artists. Since it doesn't appear at all in the expressions for $\hat{w}$, we don't need to worry about it anymore.

Resolution Comparisons

Under what conditions is the resolution of $\hat{w}$ buffering better than $\tilde{z}$ buffering? To properly compare these, we need to scale the anticipated range of $\hat{w}$ to lie between 0 and 1 just as we did for $\tilde{z}$. To do this, just set the value of d in the perspective matrix to $1/z_f$. Since $\hat{w}$ is just a scaled version of eye space z, whatever quantization steps we use for $\hat{w}$ will have the same spacing in z: equal spacing for fixed point and (approximate) logarithmic spacing for floating point.

To compare the sizes of quantization bins between the two schemes, we can compare the slopes of the depth values as functions of z. A large value of derivative is good. It means that a large variation in $\tilde{z}$ or $\hat{w}$ gives

a small variation in z. To review, we have the two competing depth functions:

$$\hat{w} = dz \qquad \tilde{z} = \frac{bz + c}{dz}$$

The derivatives of these functions are

$$\frac{d}{dz}\hat{w} = d \qquad \frac{d}{dz}\tilde{z} = \frac{-c}{dz^2}$$

So $\tilde{z}$ buffering gives better resolution if

$$\frac{-c}{dz^2} > d$$

that is, if

$$z < \frac{\sqrt{-c}}{d}$$

(Don't panic, c will always be negative.) Applying the definitions of c and d, this converts to

$$z/z_f < \sqrt{\frac{z_n/z_f}{1 - z_n/z_f}} \tag{10.2}$$

How can we interpret this? Figure 10.4 contains my visualization, showing the proper choice of $\tilde{z}$ or $\hat{w}$ buffering for all possible values of z_n. Here's how it works. Each vertical slice of the diagram stands for a choice of z_n as a proportion of z_f. We will only need to consider values of z/z_f in the shaded part of that slice, from the diagonal line (representing $z = z_n$) up to the top line (representing $z = z_f$). The curved line is a plot of the right side of Equation (10.2). If z/z_f is below this curve, we would be better off with $\tilde{z}$ buffering; if it's above, we would be better off with $\hat{w}$ buffering. Of course, you have to pick one or the other for the entire scene; this graph just tells you how much of your z range is likely to get into trouble. Finally, note that if $z_n > \frac{1}{2} z_f$, then $\tilde{z}$ buffering always wins.

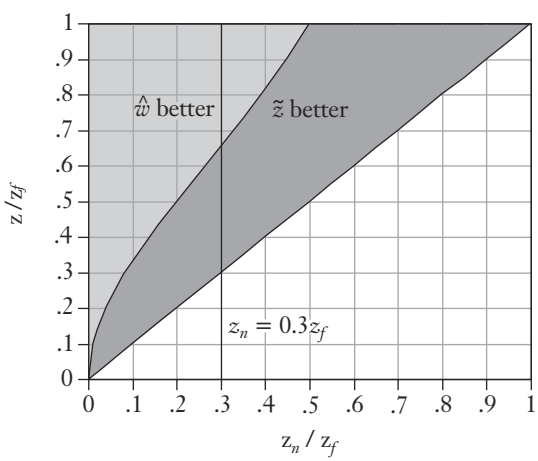

Figure 10.4 *Tradeoff between $\tilde{z}$ buffering and $\hat{w}$ buffering*

Summary

$\hat{w}$ buffering is best if you must make z_n very small, as for example, in room walk-throughs. $\tilde{z}$ buffering is best if you can get away with making z_n a noticeable fraction of z_f, as for example, in CAD, where you are examining a single object held virtually in front of you. This works well in this case mainly because the mapping for z to $\tilde{z}$ becomes pretty close to linear. The breakpoint is at about $z_n = 0.3z_f$. (See the vertical line in Figure 10.4.) If z_n is nearer than that, more of your z range would benefit from $\hat{w}$ buffering. If z_n is farther than that, more of your z range would benefit from $\tilde{z}$ buffering.

This is not the last word on the subject. We still need to investigate the desirability of using floating point versus fixed point for depth resolution. And how much resolution do you actually need? Maybe you do need more resolution for objects at small z distances because mistakes are more visually apparent when they're close to you.

Deleted Scene

Test Floating Point $1 - \tilde{z}$: Figure 10.3(b) suggests an alternative that can use the effects of floating-point quantization spacing to cancel out the nonlinearity in $\tilde{z}$: instead of calculating $\tilde{z}$, we calculate $1 - \tilde{z}$. Distant objects map to values near zero, and close objects map to values near 1 with much more evenly spread quantization steps. We, of course, have to reverse the sense of our depth comparison test in this case. Of course, we have to be careful that we aren't fooling ourselves with this. We don't want to calculate $1 - \tilde{z}$ after calculating the quantized (and information-destroyed) $\tilde{z}$. We must build the calculation into the perspective matrix, which then becomes

$$\begin{bmatrix} a & 0 & 0 \\ 0 & d-b & d \\ 0 & c & 0 \end{bmatrix}$$

and the depth value calculation becomes

$$1 - \tilde{z} = \left(\frac{z_n}{z_f - z_n} \right) \frac{z_f - z}{z}$$

I haven't pursued this.