Infini Depth: An exercise in logarithmic depth buffers

Dale Weiler

January 17, 2014

Abstract

The existing way the depth buffer is setup in hardware currently is unsuitable for modern rendering tasks where high dynamic range is paramount. There are various methods for working around the limitations imposed by the hardware depth buffer for large kilometer scale scenes; many of these methods typically involve the use of tile-based rendering and transitioning between resolutions which often leads to obvious visual artifacts. This paper describes a new method for rendering several thousands of kilometer scale scenes with near-infinite depth buffer resolution. In order to fix the problem with depth buffers we first evaluate the root of the problem to begin with; mainly, why the standard depth buffer works and why the standard depth buffer fails. We continue by explaining what the most optimal depth resolution profile is and how to better encode the the depth buffer to preserve the most optimal depth profile. We touch briefly on why normalized device coordinates in OpenGL are wrong, how we pay for a performance penalty and precision cost because of that, and why they're relevant in depth buffer precision. We then explain how logarithmic depth buffers can work on current hardware. Then finish off by explaining various optimizations we can apply to the method proposed so it's suitable.

1 Introduction

In our typical standard depth buffer setup in hardware we rely heavily on the use of a standard projection matrix which involves a near and far distance clipping plane. The projection matrix, while being entirely restriction-less in terms of how it's setup; the corresponding perspective correct depth buffer can only work if that matrix satisfies very explicit conditions. In OpenGL for instance, the value of the output w and z from a vertex shader is typically computed using

$w_p = -z$
$z_p = z(n+f)/(n-f) + 2fn/(n-f)$

This equation implies that w ends up containing the positive depth from the plane on which the camera is on. z as a result can generally be expressed as a simplified linear equation, $z_p = -az + b$. However, since w holds the positive depth, we can also express z as an even more simplified linear equation, $z_p = aw_p + b$. From here, it's quite trivial to see how the vertex shader converts these into normalized device coordinates; it's just a division by w. $z_{ndc} = a + b/w_p = a - b/z$

2 Clipping

Then those values are clipped to a range, the range of course being entirely dependent on which rendering API you're using. However hidden within all this linear algebra exists an answer to an interesting question. In the event of perspective interpolation, all vertex shader outputs are interpolated by p/w and $1/w$ then divided together to get perspective correct values, this is to give us perspective correct interpolation. Except z is never involved in this interpolation, why is that? Remember those rules the projection matrix has to satisfy for depth? Well one of those rules is that such a matrix must have a $1/z$ term in it. In our example we used $a - b/z$. That term is used by the driver to interpolate in a correct way.

So what does this have to do with depth? Well, as consequence, that value is also used as the default value for depth comparison. This default profile however isn't entirely bad, in fact it has quite interesting properties to it that make it ideal for when you need a finer resolution for near objects, with a reduced resolution for distant objects (where they get smaller in perspective). This I assume, although cannot confirm, is why I think it was chosen to begin with. However it has a problem, it wastes a lot of available range for close values. This could be fixed if hardware vendors provided a method to turn on perspective interpolation on depth components. But the likelihood of such a feature surfacing in the next iteration of GL won't happen as W-buffers are being phased out.

3 Fighting Z

The depth buffer should in the most optimal profile, provide a resolution that is proportional to the size of the geometry required to render a constant full-screen-size texture at varying distances from the camera across the entire depth range. Thus, we need a function whose derivative is proportional to $1/z$, such a function does exist, and is called the logarithm.

This $1/z$ function itself has an unfortunate property of ruining the resolution of the depth buffer, a more viable method for depth comparison would be the use of the depth values themselves. Since depth values and the depth buffer are typically floating-point to begin with, the use of floating point values would be intrinsically solid for the current depth buffer technique. The closer to the camera the closer the values get to zero. The further from the camera, the less resolution you need because the screen-size of distant objects goes down by $\approx 1/z$. More interestingly, floating-point encoding has the fortunate benefit of providing very high precision by keeping the bits in mantissa constant and only adjusting the exponent, which could be done cheaply in hardware I reckon.

Of course that doesn't really improve things does it? All we get is an increased dynamic range when we're closer to zero, which the depth buffer already provides us with; however, if we swap the near and far values we can make use of this increased range for the distant part. This almost compensates for the dwindling resolution of the $1/z$ function, but we can do better.

4 Device coordinates done wrong

OpenGL doesn't like you or me, or anybody for that matter, it actually decides to play a game of esoteric nature with normalized device coordinates that actually ends up being computationally more involved when the depth buffer is involved. In OpenGL, the normalized device coordinates are in the range $-1..1$ on all three axes including the z coordinate. The reason for this is that you typically place a camera in the center towards $+z$ or $-z$. Then as a result, z will project to the center of the screen, thus making x and y symmetric. This is actually bad for depth buffer precision, since it needs to be re-projected into depth buffer values by performing additional arithmetic $(0.5zc + 0.5)$ for the re-project.

This is annoying for a few reasons. It makes it impossible to reverse the depth range because in doing so it will only swap the mapping of near and far planes between -1 and 1. Consider the following matrix for instance

$$\begin{vmatrix} . & . & . & . \\ . & . & . & . \\ 0 & 0 & \frac{n+f}{n-f} & \frac{2fn}{n-f} \\ 0 & 0 & -1 & 0 \end{vmatrix}$$

Mapping the near plane and far plane to -1 and 1 respectfully, will leave us with something that looks vaguely similar to that of D3D

$$\begin{vmatrix} . & . & . & . \\ . & . & . & . \\ 0 & 0 & \frac{n}{n-f} & \frac{fn}{n-f} \\ 0 & 0 & -1 & 0 \end{vmatrix}$$

Oh wait, why does it map the far plane to 0 you might ask? Well matrices are funky beasts, but it's essentially a reverse-depth D3D projection matrix and it has an inverted sign, which means the depth function doesn't actually change at all, but we can switch that to 1..0 range later. The important thing is that clipping still applies at 1..0 so we have to use a custom clipping plane to clip at 0. But we can ignore all of this if we ensure our geometry never needs to be cliped behind the far plane. This works actually surprisingly well, but we can do better.

Remember the re-projection of depth buffer values from earlier? Well that additive term in the remapping ends up causing some issues. It locks the floating-point exponent and it destroys all additional precision that our clever encoding method brings for values close to zero. So as a result, we're stuck with 24 bits of mantissa to handle that $1/z$. However, Nvidia has provided us with a method to eliminate that bias, and it's even present on ATI as of Catalyst 13.11. The depth buffer float extension gives us glDephRangeNV, a single call of that with -1,1 will remove all that bias and will essentially give us D3D-style mapping, dodged a bullet there. I should note that the use of glDepthRange cannot be used here as the OpenGL spec has a note saying that the arguments will be clamped to 0..1 range, that's no good.

5 Logarithms to the rescue

We can use the ideal logarithmic distribution described earlier to solve some of these problems, we can also do it on current hardware, for free.

The technique for a logarithmic depth buffer works by outputting some desired logarithmic value from the vertex shader, pre-multiplied by the w value to eliminate perspective divide. It's quite easy to do, assuming you have near and far clipping plane varyings at your disposal.

```
// near = near clipping plane
// far = far clipping plane
// coef = coefficent for linear part of depth mapping function
#define ZCO(OP, NEAR) 2.0 * log(gl_Position OP NEAR) / log(far OP NEAR) - 1
#ifdef CONSTANT_RELATIVE_PRECISION
# define ZDEPTH ZCO(*, coef + 1)
#else
# define ZDEPTH CZO(/, near)
#endif

gl_Position.z = ZDEPTH;
gl_Posititn.z *= gl_Position.w; // eliminate perspective divide
```

This typically produces adequate precision across the entire depth range with some left over even, but it has some issues with long polygons which are close to the camera. This is what happens when you use a vertex shader, the values are only ever correct at the vertices. The problem is interpolated values at pixels often aren't the same because of two reasons.

1. Non linearity in the logarithmic function between two depth values
2. Implicit linear interpolation of depth value in the rasterizer

Both of these reasons can be solved quite trivially by writing the correct values to gl_FragDepth in the fragment shader, but it's also the wrong solution, as it increases bandwidth when stencil is needed and breaks any depth-related optimizations the hardware/driver provides.

6 Logarithmic Depth Buffers

So we know how to do it, but how do we do it for free?, after all we did make such a claim earlier did we not? Well, lets first establish how such exact-pixel value of the logarithmic depth is computed. In essence we know interpolation of the depth is involved and the logarithm can be computed in the fragment shader, we also know that the problem of depth interpolation itself only appears for close objects. What if we could linearize the logarithmic curve for just the region that is close to the camera? This way we can simply output the interpolant without requiring any expensive operations in the fragment shader.

Recall from the earlier sample shader the idea of a coefficient for constant relative precision, well, that's essentially the solution. By adjusting that coefficient we can actually change the width of the linear part of the depth mapping function. Through various experimenting I've determined that a coefficient of 0.01 is sufficient for 10 meters (that of a typical FPS).

Provided for reference is an equation that can help you determine that coefficient, resolution of some distance x, for a given $coeff$ and n bits of Z-buffer resolution can be calculated as.

$$distance = log(coeff * far + 1)/((2^n - 1) * coeff/(coeff * x + 1)$$

So for example, a far plane for $10k$ km distance and a 24-bit Z-Buffer gives a table of the following resolutions

	1m	10m	100m	1km	10km	100km
coeff=1	1.9e-2	1.1e-5	9.7e-5	0.001	0.01	0.096
coeff=0.001	0.0005	0.0005	0.0006	0.001	0.006	0.055

6.1 Integration

To integrate this, changes need to be made to the vertex shader to allow outputting logarithmic Z. The final shader set will look something like this

```
// vertex
out float log_z;

// post projection
const float coeff_far = 1.0/log(far * coeff + 1;
log_z = log(gl_Position.w * coeff + 1) * coeff_far;
gl_Position.z = (2 * log_z - 1) * gl_Position.w;

// fragment
in float log_z;
void main() {
   // ...
   gl_FragDepth = log_z;
   // ...
}
```

6.2 Being conservative

As we know, writing directly to gl_FragDepth completely disables any early-z optimizations. This would typically be very bad, except that we can still improve further. Logarithmic functions are monotonous, which means we can use ARB_conservative_depth to provide a hint to the driver that our depth values from the fragment shader will always be well below or above the interpolated ones. In doing so, this will allow the driver to skip all evaluation for fragments that are discarded.

```
#extension GL_ARB_conservative_depth : enable
layout(depth_less) out float gl_FragDepth;
```

What, depth_less, Surely that's wrong, after all, the secant on our logarithmic curve always lies below it, right? No, actually, while the value of z is being interpolated linearly in the *rasterizer*, the interpolant which is used to set gl_FragDepth is interpolated *perspectively* by interpolating p/w and $1/w$ *linearly* and then getting the correct value in the process of *dividing* the two. They aren't the same, it becomes quite clear as a comparison between the following

$((1-t)*log(A+1)+t*log(B+1))$
$((1-t)*log(A+1)/A+t*log(B+1)/B)/((1-t)/A+t/B)$

Which means that perspectively interpolated values always go below the linearly interpolated ones. With depth comparison function GL_LESS, values written to gl_FragDepth can only ever be close to the camera. Of course this doesn't matter because it's of no use for early-z rejection. But it does help.

6.2.1 Adaptive tesselation helps

For large world geometry such as terrains, mountains, grass and other large models, it's actually cheaper to have them tesselated adaptively. The reason for this is because of the explicit writing of depth values in the fragment shader. While we may not benefit entirely from all the optimizations considered thus far, we can reclaim some of it back by having dynamic control over fragment depth writes on individual objects which are rendered. From here it could also be beneficial to use this method of dynamic tesselation for other objects that cross some a depth range threshold.

7 Conclusion

Logarithmic depth buffer with a couple of optimizations can make infinite-depth possible on current hardware assuming a 24-bit Z. While we may lose early-z and require that geometry never be clipped behind the far plane we consider these sensible compromises. In the case that stencil is used we consider the bandwidth requirements of logarithmic depth buffers less than ideal, but ultimately unsolvable.

The correct solution would be to implement hardware support for optimal depth buffer utilization such that the bandwidth can be reduced over all. We also consider that the option to enable perspective interpolation on z component with linear or logarithmic depth without any loss of performance in hardware would be useful as well.