An Efficient Depth Linearization Method for Oblique View Frustums

Alexander V. Popov
Novosibirsk State University

Abstract

Many modern 3D rendering techniques operate on fragment depths and require the sampling of depth textures, containing non-linear window space z-coordinate values, for depth values of certain fragments. In most cases, such values must be linearized before use, in particular to compare fragment depths, to perform view space arithmetic, or to compare depths of scenes rendered with different projections. Transforming depth from window space to view space may become complicated and unintuitive in the case of an oblique frustum with a modified near clipping plane that is not parallel to a conventional near plane. Such frustums can be utilized for effective and high-performance geometry clipping to a user-defined plane. This paper discusses an efficient depth value linearization method for an arbitrary projection matrix, which can be used in post-processing, ambient occlusion, and other depth-dependent effects.

1. Introduction

Many modern 3D rendering techniques, including, but not limited to ambient occlusion, full scene anti-aliasing, depth of field, and others, operate on fragment depths and require the sampling of depth values of certain fragments. Such sampling is performed using depth textures, by copying the depth buffer contents or by directly rendering depth into them. Sampled depth values (window space z-coordinates) are non-linear due to the nature of perspective projection [Segal and Akeley 2016] and, in most cases, must be linearized before use. It is common to use an equation involving near and far clipping-plane distances for such linearization, which is fairly simple and fast and takes the form of

$$z_v = \frac{1}{Az_w + B}, \tag{1}$$

where z_v is a view space z-coordinate, z_w is a window space z-coordinate (stored in the depth buffer or depth texture), and A and B are constants, passed to a frag-

 ISSN 2331-7418

ment program; the constants depend on current z-near and z-far values of a current projection.

Geometry clipping is another important problem, in particular, for multi-pass rendering engines. Such visual effects as portals and mirrors are usually implemented using additional, possibly recursive, passes, drawing a scene from camera positions different than the primary one. The geometry rendered in additional passes must somehow be clipped to a mirror or portal plane, thus eliminating unwanted penetration artifacts. There are several user-defined clipping approaches, but one of the most clever and efficient is utilizing the hardware near clipping plane of the frustum [NVIDIA 2003], placing it in such a way that it coincides with the portal plane. However, such oblique frustum modification effectively destroys the far clipping plane [Lengyel 2005], resulting in near and far clipping distances becoming nonsense.

This article presents a simple and optimal solution for linearization of such "skewed" depth values into exact view space z-coordinates, derived directly from the projection matrix transformation, so the clipping-plane orientation does not matter. Diminished depth buffer precision, apparently the only issue of oblique frustum clipping, has been proven to be manageable for 24-bit depth buffers [Lengyel 2005]. The solution, compatible with common projection matrices, only requires adding one division operation to a fragment program, two scalar vertex interpolants, and a couple of instructions to a vertex program.

1.1. Related Work

User-defined clipping planes become very important in multi-pass rendering techniques that, by their nature, require geometry to be clipped to an arbitrary plane: portals, mirrors, reflective water surface, and so on. If the clipping is not performed, some geometry penetrates through the physical border of the portal surface and results in visual artifacts. There are several options available to enable user-defined clipping planes on modern hardware.

First, both Direct3D and OpenGL provide API functions to set up and enable such planes in addition to six conventional frustum planes. While this approach works well in fixed-function pipelines, it is inconvenient within programmable pipelines, which are likely to be used by most modern rendering software. On older GPUs, such clipping may result in software vertex processing [Advanced Micro Devices 2007], thus severely dropping frame rate. In addition it requires vertex shaders to output additional variables (distances to clipping planes), resulting in duplicated shader sets: one for regular rendering and another for rendering with the clipping plane enabled. Second, clipping can be performed manually in shaders, discarding fragments that are on a certain side of the clipping plane. Unfortunately, explicit fragment discarding in a fragment program may disable some vital optimizations (e.g., Early-Z) [NVIDIA 2008] and, again, lead to poor performance. Tile-based mobile GPUs are extremely

vulnerable to the problem and must minimize explicit discards in fragment programs as much as possible [Smedberg 2012; Merry 2012].

Another interesting technique, "oblique frustum clipping," has been developed to resolve these issues. It utilizes one of the six hardware clipping planes, so the user-defined clipping is performed at no cost [NVIDIA 2003; Lengyel 2005]. The main idea is to modify a standard projection matrix in such a way that the near clipping plane coincides with the user-defined plane. The latter is achieved by replacing the third row of a projection matrix. Although such a modification has a detrimental effect on the conventional far clipping plane [Lengyel 2005], this issue has been resolved by Lengyel, so that the depth buffer precision is kept reasonable. The third row of a new, oblique projection matrix has the form

$$\vec{P}_3 = \frac{2(\vec{P}_4 \cdot \vec{Q})}{\vec{C} \cdot \vec{Q}} \vec{C} - \vec{P}_4,$$

where $\vec{P}_i$ is an ith row of the projection matrix $\mathbf{P}$, $\vec{C}$ is a user-defined clipping plane equation in view space, and $\vec{Q}$ is a view space corner of the frustum opposite to the plane $\vec{C}$. The same approach has been later extended to orthographic projection matrices [Pranckevičius 2007].

It is easy to notice that the third row of a projection matrix now has all four components non-zero, which introduces dependence of the clip space z-coordinate, not only on view space z, far and near plane distances, but also on view space x- and y-coordinates. This makes depth linearization unintuitive and more complicated, and this problem is addressed in the present work.

2. Method Overview

Both regular and oblique projection matrices are invertible by design [Lengyel 2005], which means that we have a bijective relation between view space and window space coordinates. Therefore, we can apply an inverse projection transformation to clip space coordinates to get view space coordinates, and in particular view space z. The problem is that none of the clip space coordinates are known. The only value we have is a window space z-coordinate, which can be directly sampled from a depth texture. Assuming the default depth range (0–1) and clip control (negative-one-to-one), this yields the OpenGL normalized device space z_d:

$$z_d = 2z_w - 1. \tag{2}$$

In Direct3D [Microsoft], and in OpenGL with zero-to-one clip control enabled [Segal and Akeley 2016], scale and bias in Equation (2) are 1.0 and 0.0, respectively, and can be omitted.

Given the device space z_d, we can define an equation (Equation (3)) containing view space z_v:

$$z_d = \frac{z_c}{w_c} = \frac{\vec{v} \cdot \vec{P}_3}{\vec{v} \cdot \vec{P}_4} = \frac{x_v P_{31} + y_v P_{32} + z_v P_{33} + w_v P_{34}}{x_v P_{41} + y_v P_{42} + z_v P_{43} + w_v P_{44}}, \tag{3}$$

where $\vec{v}$ is a view space position of the fragment, $\vec{P}_i$ is an ith row of the projection matrix $\mathbf{P}$, and P_{ij} is a jth component of the ith row of the projection matrix $\mathbf{P}$. Since most projection matrices place a negated z_v-component to the w component of the transformed vertex for perspective correctness, the 4th row of the projection matrix is usually $(0, 0, -1, 0)$. Taking the latter into consideration and assuming $w_v = 1$, Equation (3) simplifies to

$$z_d = \frac{z_c}{w_c} = -\frac{x_v P_{31} + y_v P_{32} + z_v P_{33} + P_{34}}{z_v}. \tag{4}$$

The only problem is that we still don't have x_v and y_v values. But since a depth texture is usually sampled in a projective manner, i.e., at the same x- and y-coordinates where the current fragment is located, we can easily calculate them. The main idea is that since window space x_w and y_w are the same, normalized device x_d and y_d are also the same. Thus, we can pass clip space coordinates from a vertex shader and, after perspective division, we get x_d and y_d of the current fragment, and, luckily, of the sampled fragment. This is true because

$$x_d = \frac{x_c}{w_c} = \frac{\vec{v} \cdot \vec{P}_1}{\vec{v} \cdot \vec{P}_4} = -\frac{x_v P_{11} + y_v P_{12} + z_v P_{13} + w_v P_{14}}{z_v},$$
$$y_d = \frac{y_c}{w_c} = \frac{\vec{v} \cdot \vec{P}_2}{\vec{v} \cdot \vec{P}_4} = -\frac{x_v P_{21} + y_v P_{22} + z_v P_{23} + w_v P_{24}}{z_v},$$

and assuming the frustum is symmetric in both horizontal and vertical directions, we have

$$x_d = \frac{x_c}{w_c} = -\frac{x_v P_{11}}{z_v},$$
$$y_d = \frac{y_c}{w_c} = -\frac{y_v P_{22}}{z_v}.$$

So we can express x_v and y_v by x_d, y_d and z_v:

$$x_v = -\frac{x_d}{P_{11}} z_v,$$
$$y_v = -\frac{y_d}{P_{22}} z_v.$$

Finally, we substitute these values into Equation (4) and solve it for z_v, resulting in

$$z_v = \frac{1}{x_d \frac{P_{31}}{P_{11} P_{34}} + y_d \frac{P_{32}}{P_{22} P_{34}} - \frac{1}{P_{34}} z_d - \frac{P_{33}}{P_{34}}}. \tag{5}$$

If the frustum is not symmetric, Equation (5) becomes slightly more complicated (we'll have to take into account non-zero P_{13}- and P_{23}-components of the projection matrix), but still solvable for z_v. Indeed, normalized device coordinates x_d and y_d become

$$x_d = \frac{x_c}{w_c} = -\frac{x_v P_{11}}{z_v} - P_{13},$$
$$y_d = \frac{y_c}{w_c} = -\frac{y_v P_{22}}{z_v} - P_{23},$$

which, after evaluation of x_v and y_v, subsequent substitution into Equation (4), and solving for z_v in the same way, yields the generalized equation for z_v:

$$z_v = \frac{1}{x_d \frac{P_{31}}{P_{11}P_{34}} + y_d \frac{P_{32}}{P_{22}P_{34}} - \frac{1}{P_{34}} z_d - \left(\frac{P_{33}}{P_{34}} - \frac{P_{31}P_{13}}{P_{11}P_{34}} - \frac{P_{32}P_{23}}{P_{22}P_{34}} \right)} \tag{6}$$

Speaking about arbitrary projection matrices, we cannot but mention view space z decoding for orthographic projections. Although the term "linearization" is not applicable here due to the linear nature of the depth values, we still need to calculate z_v accurately. Taking the form of the orthographic projection matrix into account, and again assuming w_v = 1, we have

$$z_v = -x_d \frac{P_{31}}{P_{11}P_{33}} - y_d \frac{P_{32}}{P_{22}P_{33}} + \frac{1}{P_{33}} z_d - \left(\frac{P_{34}}{P_{33}} - \frac{P_{31}P_{14}}{P_{11}P_{33}} - \frac{P_{32}P_{24}}{P_{22}P_{33}} \right). \tag{7}$$

3. Optimization

A further optimization for our method is to determine which parts of Equation (5) or Equation (6) can be calculated in the vertex program and passed to the fragment program. We want to keep calculations at fragment level simple, maintaining a common form of zv as shown in Equation (1), but with A and B not necessarily constant. This basically requires one multiply-add and one reciprocal instruction, which will be the same as we would use to decode standard depth. But since B is a function of x and y and is actually an interpolated value, we have to perform an additional (perspective) division at fragment level. Let our projection matrix be symmetric for simplicity, and use Equation (5). In a vertex program, we compute a varying 2-component vector $\vec{B}$:

$$\vec{B} = \left(x_c \frac{P_{31}}{P_{11}P_{34}} + y_c \frac{P_{32}}{P_{22}P_{34}} - w_c \frac{P_{33}}{P_{34}},\ w_c \right)$$

and pass a uniform scalar A, along with scale and bias to map z_v to a proper interval, if any. We also pass a 4-component uniform vector $\vec{U}_v$ to a vertex program from the CPU (negated, to make z_v positive):

$$\vec{U}_v = \left(-\frac{P_{31}}{P_{11}P_{34}},\ -\frac{P_{32}}{P_{22}P_{34}},\ 0,\ \frac{P_{33} + \text{bias}}{P_{34}} \right) \tag{8}$$

where *bias* depends on a range of a normalized device z_d, i.e., on a clip control (see Equation (2)). In Direct3D, and in OpenGL with zero-to-one clip control enabled, bias is 0, and in OpenGL with negative-one-to-one clip control (the default value) it is -1. For that reason, we should also apply scale to the value of A passed from the CPU to a fragment program as a U_f uniform floating-point scalar:

$$U_f = -\frac{\text{scale}}{P_{34}} \tag{9}$$

where scale is either 1.0 or 2.0, respectively.

For a generalized off-center perspective projection matrix case, the bias should also contain additional constant components from Equation (6).

The final shader code is shown in Listing 1. Uniform parameters $\vec{U_v}$ and U_f are set as defined by Equations (8) and (9). The resulting z_v is perspective projection-matrix–independent and clearly represents the eye space distance in view coordinates. It can further be scaled and biased to map into an appropriate range, e.g., relative distance between near and far clipping planes (note that for oblique projections, these are actually nominal values) and used in depth comparisons. The same approach can be used to decode z_v in orthographic projections, with appropriate modifications of constants according to Equation (7) and elimination of the reciprocal operation in the fragment program.

The method described above is also valid for standard projections without any near clipping plane modifications. In this case, $P_{31} = P_{32} = 0$, and Equation (5) yields

$$z_v = \frac{1}{-\frac{1}{P_{34}} z_d - \frac{P_{33}}{P_{34}}}$$

where P_{33} and P_{34} are expressed only with near and far clipping plane distances, and both A and B are constants for a certain projection matrix.

```
VERTEX PROGRAM:
uniform float4 Uv; // see Equation (8)
...
output.position = mul( input.position, ModelviewProjectionMatrix );
output.b.x = dot( output.position, Uv );
output.b.y = output.position.w;
...
FRAGMENT PROGRAM:
uniform float Uf; // see Equation (9)
...
float zw = tex2Dproj( depthSampler, input.projCoords.xyw ).r;
float zv = 1.0 / ( zw * Uf + input.b.x / input.b.y );
...
```

Listing 1. Shader code for frustum computation.

4. Conclusion

We have presented an efficient method of linear view space depth reconstruction to be used with oblique frustum clipping. The approach works well with all popular projection types, including centered and off-screen perspective projections; the orthographic case is also addressed. Originally developed for our own 3D game engine (Volatile Engine), the method is presented in the hope that it will be useful for the wider community of developers. The applications of the algorithm may include per-pass post-processing effects, view space depth dependent effects (SSAO, HBAO), anti-aliasing, water edge smoothing, and others.

References

ADVANCED MICRO DEVICES, I. 2007. *ATI OpenGL Programming and Optimization Guide*. URL: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/ATI_OpenGL_Programming_and_Optimization_Guide.pdf. 37

LENGYEL, E. 2005. Oblique view frustum depth projection and clipping. *Journal of Game Development 1*, 2, 5–16. URL: http://www.terathon.com/lengyel/Lengyel-Oblique.pdf. 37, 38

MERRY, B. 2012. Performance tuning for tile-based architectures. In *OpenGL Insights*, P. Cozzi and C. Riccio, Eds. CRC Press, Natick, MA. URL: http://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-TileBasedArchitectures.pdf. 38

MICROSOFT. *The Direct3D Transformation Pipeline*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ee418867(v=vs.85).aspx. 38

NVIDIA, 2003. Oblique frustum clipping. Online source code. URL: http://www.nvidia.com/object/oblique_frustum_clipping.html. 37, 38

NVIDIA. 2008. *GeForce 8 and 9 Series GPU Programming Guide*. NVIDIA Corporation. URL: http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide_G80.pdf. 37

PRANCKEVIČIUS, A., 2007. Oblique near-plane clipping with orthographic camera. Online. URL: http://aras-p.info/texts/obliqueortho.html. 38

SEGAL, M., AND AKELEY, K. 2016. *The OpenGL Graphics System: A Specification*. The Khronos Group. URL: https://www.opengl.org/registry/doc/glspec45.core.pdf. 36, 38

SMEDBERG, N. 2012. Bringing AAA graphics to mobile platforms. In *Proceedings of the 2012 Game Developers Conference*. URL: http://www.gdcvault.com/play/1015331/Bringing-AAA-Graphics-to-Mobile. 38

Author Contact Information

Alexander V. Popov
Novosibirsk State University
Novosibirsk, Russia, 630090
apopov@niboch.nsc.ru

Alexander V. Popov, An Efficient Depth-Linearization Method for Oblique View Frustums, *Journal of Computer Graphics Techniques (JCGT)*, vol. 5, no. 4, 36–43, 2016
http://jcgt.org/published/0005/04/03/

Received:	2016-07-06		
Recommended:	2016-08-24	Corresponding Editor:	Naty Hoffman
Published:	2016-12-20	Editor-in-Chief:	Marc Olano