To appear in Eurographics 2013

Stochastic Depth Buffer Compression using Generalized Plane Encoding

M. Andersson[†1,2], J. Munkberg[1] and T. Akenine-Möller[1,2]

[1]Intel Corporation [2]Lund University

Abstract

In this paper, we derive compact representations of the depth function for a triangle undergoing motion or defocus blur. Unlike a static primitive, where the depth function is planar, the depth function is a rational function in time and the lens parameters. Furthermore, we show how these compact depth functions can be used to design an efficient depth buffer compressor/decompressor, which significantly lowers total depth buffer bandwidth usage for a range of test scenes. In addition, our compressor/decompressor is simpler in the number of operations needed to execute, which makes our algorithm more amenable for hardware implementation than previous methods.

Categories and Subject Descriptors (according to ACM CCS): E.4 [Coding and Information Theory]: Data compaction and compression—I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

1. Introduction

Depth buffering is the standard technique to resolve visibility between objects in a rasterization pipeline. A *depth buffer* holds a depth value for each sample, representing the current closest depth of all previously rendered triangles overlapping the sample. In a stochastic rasterizer with many samples per pixel, the depth buffer bandwidth requirements are much higher than usual, and the depth data should be compressed if possible. The depth value, d, can be defined in a number of ways. In current graphics hardware APIs, the normalized depth, $d = \frac{z_{clip}}{w_{clip}}$, is used since it is bounded to $[0,1]$, and distributes much of the resolution closer to the viewer. Alternatively, the raw floating-point value, w_{clip}, can be stored. The former representation has the important property that the depth of a triangle can be linearly interpolated in screen space, which is exploited by many depth buffer compression formats. Unfortunately, for moving and defocused triangles, this is no longer true. Therefore, we analyze the mathematical expression for the depth functions in the case of motion blur and depth of field. We show that although the expressions may appear somewhat complicated, they can be effectively simplified, and compact forms for the depth functions can be used to design algorithms with substantially better average compression ratios for stochastic rasterization. Our method only targets motion *or* defocus blur, but for completeness, we also include the derivation for simultaneous motion blur and depth of field in Appendix A.

In general, we assume that the compressors and decompressors exist in a depth system, as described by Hasselgren and Akenine-Möller [HAM06]. Compression/decompression is applied to a *tile*, which typically is the set of depth samples inside a rectangular screen-space region. Due to bus widths, compression algorithms, and tile sizes, only a few different compression ratios, e.g., 25% & 50%, are usually available. Typically, a few bits (e.g., two) are stored on-chip, or in a cache, and these are used to indicate the compression level of a tile, or whether the tile is uncompressed or in a fast clear mode.

2. Previous Work

Previous depth compression research targeting static geometry typically exploits that the depth function, $d(x,y) = \frac{z(x,y)}{w(x,y)}$, is linear in screen space, (x,y), and this can be used to achieve high compression ratios. Morein was the first to describe a depth compression system, and he used a differential-differential pulse code modulation (DDPCM) method for compression [Mor00]. By examining patent data bases on depth compression, Hasselgren and Akenine-

† magnusa@cs.lth.se

Möller presented a survey on a handful of compression algorithms [HAM06].

One of the most successful depth compression algorithms is *plane encoding* [HAM06], where the rasterizer feeds the exact plane equations to the compressor together with a coverage mask indicating which samples/pixels inside a tile that are covered by the triangle. The general idea is simple. When a triangle is rendered to a tile, first check whether there is space in the compressed representation of the tile for another triangle plane equation. If so, we store the plane equation, and update the plane-selection bit mask of the tile to indicate which samples point to the new plane equation. When there is not enough space to fit any more plane equations, we need to decompress the current depths, update with the new incoming depth data, and then store the depth in an uncompressed format. To decompress a tile, just loop over the samples, and look at the plane selection bit mask to obtain the plane equation for the sample, and evaluate that plane equation for the particular, (x,y), of the sample. A compressed tile will be able to store n plane equations together with a plane selection bit mask with $\lceil \log n \rceil$ bits per sample, where n depends on the parameters of the depth compression system, and the desired compression ratio. In this paper, we generalize this method so that it works for motion blur and defocus blur, and we optimize the depth function representations.

Anchor encoding is a method similar to plane encoding. It uses approximate plane equations (derived from the depth data itself, instead of being fed from the rasterizer), and encodes the differences, also called residuals, between each depth value and the predicted depth from the approximate plane equation. *Depth offset* encoding is probably one of the simplest methods. First, the minimum, Z_{min}, and the maximum, Z_{max}, depths of a tile are found. Each sample then uses a bit to signal whether it is encoded relative to the min or the max, and the difference is encoded using as many bits that are left to reach the desired compression ratio.

The first public algorithm for compressing floating-point depth [SWR*08] reinterpreted the floats as integers, and used a predictor based on a small set of depths in the tile. The residuals, i.e., the difference between the predicted values and the real depths, were then entropy encoded using Golomb-Rice encoding. A general method for compressing floating-point data was presented by Pool et al. [PLS12]. The differences between a sequence of floating-point numbers is encoded using an entropy encode based on Fibonacci codes.

A compressed depth cache was recently documented [HANAM13], and some improvements to depth buffering were described. In particular, when data is sent uncompressed, smaller tile sizes are used compared to when the tiles are compressed. We will also use this feature in our research presented here.

Color buffer compression algorithms [RHAM07, SWR*08, RSAM08] are working on different data (color instead of depth), but otherwise, those algorithms operate in a similar system inside the graphics processor as do depth compression.

Gribel et al. [GDAM10] perform lossy compression of a time-dependent depth function per pixel. However, this approach requires a unique depth function per pixel, and does not solve the problem of compressing stochastically generated buffers over a tile of pixels. They derive the depth function for a motion blurred triangle and note that when the triangle is moving, the linearity of the depth in screen space is broken. From their work, we know that the depth is a rational cubic function of time, t, for a given sample position (x,y).

Recently, higher-order rasterization, i.e., for motion blur and depth of field, has become a popular research topic. All existing methods for static geometry, except depth offset encoding, break down in a stochastic rasterizer [AMMH07, AHAM11]. In the depth compression work by Andersson et al. [AHAM11], the time dimension was incorporated in the compression schemes, which led to improved depth buffer compression for stochastic motion blur. By focusing on both motion blur and defocus blur, we solve a much larger problem. In addition, we analyze the depth functions and simplify their representations into very compact forms.

3. Background

In this section, we give some background on barycentric interpolation and show how the depth function, $d = \frac{z}{w}$, is computed for static triangles. Towards the end of this section, we show a generalized version of the depth function without deriving the details. All this information is highly useful for the understanding of the rest of the paper.

Suppose we have a triangle with clip space vertex positions $\mathbf{p}_k = [p_{k_x}, p_{k_y}, p_{k_w}]$, $k \in \{0,1,2\}$. In homogeneous rasterization, the 2D homogeneous (2DH) edge equation, $e_k = \mathbf{n}_k \cdot \mathbf{x}$, corresponds to a distance calculation of an image plane position, $\mathbf{x} = [x,y,1]^T$, and the edge plane, which passes through the origin, with, for example, $\mathbf{n}_2 = \mathbf{p}_0 \times \mathbf{p}_1$.

Let us introduce an arbitrary per-vertex attribute, A_k, that we wish to interpolate over the triangle. McCool et al. [MWM02] showed that each of the barycentric coordinates, B_0, B_1, B_2, of the triangle can be found by evaluating and normalizing the corresponding 2DH edge equation, such that $B_k = \frac{e_k}{e_0+e_1+e_2}$. The interpolated attribute, A, for a given sample point, $\mathbf{x}$, can then be found by standard barycentric interpolation:

$$A(x,y) = \sum_{k=0}^{2} A_k B_k = \frac{A_0 e_0 + A_1 e_1 + A_2 e_2}{e_0 + e_1 + e_2}. \qquad (1)$$

The depth value, d, is formed by interpolating z and w individually, and then performing a division:

$$d(x,y) = \frac{z(x,y)}{w(x,y)} = \frac{\sum z_k B_k}{\sum w_k B_k} = \frac{\sum z_k e_k}{\sum w_k e_k}. \qquad (2)$$

If we look at the denominator, we see that:[†]

$$\sum_{k=0}^{2} w_k e_k = (\sum_{k=0}^{2} w_k \mathbf{p}_i \times \mathbf{p}_j) \cdot \mathbf{x} \quad (3)$$
$$= [0,0,\det(\mathbf{p}_0,\mathbf{p}_1,\mathbf{p}_2)] \cdot \mathbf{x} = \det(\mathbf{p}_0,\mathbf{p}_1,\mathbf{p}_2),$$

which is independent of (x,y). This is six times the signed volume of the tetrahedron spanned by the origin and the triangle, which can be used to detect if a triangle is backfacing.

If we use a standard projection matrix, such that the transformation of $(z_{cam},1)$ to clip space (z,w) can be expressed as (c.f., the standard Direct3D projection matrix):

$$z = a\, z_{\text{cam}} + b, \quad w = z_{\text{cam}}, \quad (4)$$

then the depth function can be simplified. The coefficients a and b depend solely on z_{near} and z_{far}. Combining Equations 2 and 4 and simplifying gives us:

$$d(x,y) = \frac{z(x,y)}{w(x,y)} = a + \frac{b\sum e_k}{\sum w_k e_k}. \quad (5)$$

We have now derived the 2D depth function, which is widely used in rendering systems today. However, Equation 5 can be augmented so that it holds for depth sampled in higher dimensions. For example, adding motion blur and depth of field means that z, w, and the edge equations are functions of shutter time, t, and lens position, (u,v). Thus, we can write the depth function on a more general form:

$$d(x,y,\ldots) = a + \frac{b\sum e_k(x,y,\ldots)}{\sum w_k(x,y,\ldots)e_k(x,y,\ldots)}, \quad (6)$$

where $\ldots$ should be replaced with the new, augmented dimensions.

4. Generalized Plane Encoding

In Section 2, we described how the plane encoding method works for static triangle rendering. For higher-order rasterization, including motion blur and defocus blur, static plane equations cannot be used to represent the depth functions, because the depth functions are much more complex in those cases. For motion blur, the depth function is a cubic rational polynomial [GDAM10], for example. Therefore, the goal of our work in this paper is to generalize the plane encoding method in order to also handle motion blur and defocus blur.

Our new *generalized plane encoding* (GPE) algorithm is nearly identical to static plane encoding, except that the plane equations for motion blurred and/or defocused plane equations use more storage, and that the depth functions are more expensive to evaluate. This can be seen in Equation 6, which is based on more complicated edge equations, e_k, and w_k-components. However, in Section 5, we will show how the required number of coefficients for specific cases can be substantially reduced, which makes it possible to fit more planes in the compressed representation. This in turn makes for higher compression ratios and faster depth evaluation.

Similar to static plane encoding, the compression representation for generalized depth (motion and defocus blur, for example) includes a variable number of generalized plane equations (Section 5), and a plane selector bitmask per sample. If there are at most n plane equations in the compressed representation, then each sample needs $\lceil \log n \rceil$ bits for the plane selector bitmask. Next, we simplify the depth functions for higher-order rasterization.

5. Generalized Depth Function Derivations

In the following subsections, we will derive compact depth functions for motion blurred and defocused triangles. Some readers may want to skip to the results in Section 7. Since we ultimately could not simplify the combined defocus and motion blur case, we skip that derivation in this section and refer interested readers to Appendix A.

5.1. Motion Blur

We begin the depth function derivation for motion blur by setting up time-dependent attribute interpolation in matrix form. Then, we move on to reducing the number of coefficients needed to exactly represent the interpolated depth of a triangle.

The naïve approach to store the depth functions for a motion blurred triangle is to retain all vertex positions at $t = 0$ and $t = 1$, which are comprised of a total of $4 \times 3 \times 2 = 24$ coordinate values (e.g., floating-point). If the projection matrix is known, and can be stored globally, then only $3 \times 3 \times 2 = 18$ coordinate values are needed, as z then can be derived from w. In the following section, we show how the depth function can be rewritten and simplified to contain only 13 values, which enables more efficient storage.

Time-dependent Barycentric Interpolation In the derivation below, we assume that vertices move linearly in clip space within each frame. Thus, the vertex position, $\mathbf{p}_k$, becomes a function of time:

$$\mathbf{p}_k(t) = \mathbf{q}_k + t\mathbf{d}_k, \quad (7)$$

where $\mathbf{d}_k$ is the corresponding motion vector for vertex k. Akenine-Möller et al. [AMMH07] showed that since the vertices depend on time, the 2DH edge equations form 2nd degree polynomials in t:

$$e_k(x,y,t) = (\mathbf{p}_i(t) \times \mathbf{p}_j(t)) \cdot \mathbf{x} = (\mathbf{h}_k + \mathbf{g}_k t + \mathbf{f}_k t^2) \cdot \mathbf{x}, \quad (8)$$

where

$$\mathbf{h}_k = \mathbf{q}_i \times \mathbf{q}_j, \;\; \mathbf{g}_k = \mathbf{q}_i \times \mathbf{d}_j + \mathbf{d}_i \times \mathbf{q}_j, \;\; \mathbf{f}_k = \mathbf{d}_i \times \mathbf{d}_j. \quad (9)$$

For convenience, we rewrite the edge equation in matrix form:

[†] Throughout the paper, we will sum over k, $k \in \{0,1,2\}$ and use the notation $i = (k+1) \bmod 3$ and $j = (k+2) \bmod 3$.

$$e_k(x,y,t) = \mathbf{t}_2\mathbf{C}_k\mathbf{x}, \text{ where } \mathbf{C}_k = \begin{bmatrix} h_{kx} & h_{ky} & h_{kw} \\ g_{kx} & g_{ky} & g_{kw} \\ f_{kx} & f_{ky} & f_{kw} \end{bmatrix}, \tag{10}$$

where $\mathbf{t}_2 = [1, t, t^2]$ is a row vector and $\mathbf{x} = [x, y, 1]^{\mathrm{T}}$ is a column vector. By combining the matrix notation and Equation 1, we have a general expression of how to interpolate a vertex attribute, A_k, over a motion blurred triangle:

$$A(x,y,t) = \frac{\mathbf{t}_2(\sum A_k\mathbf{C}_k)\mathbf{x}}{\mathbf{t}_2\sum\mathbf{C}_k\mathbf{x}}. \tag{11}$$

However, if the attribute itself varies with t, e.g., $A_k(t) = A_k^o + tA_k^d$, we obtain a general expression for interpolating a time-dependent attribute over the triangle, with a numerator of cubic degree:

$$A(x,y,t) = \frac{\mathbf{t}_2\sum((A_k^o + tA_k^d)\mathbf{C}_k)\mathbf{x}}{\mathbf{t}_2\sum\mathbf{C}_k\mathbf{x}} = \frac{\mathbf{t}_3\mathbf{C}_A\mathbf{x}}{\mathbf{t}_2\sum\mathbf{C}_k\mathbf{x}}, \tag{12}$$

where $\mathbf{t}_3 = [1, t, t^2, t^3]$, and the vertex attributes, A_k, are multiplied with each $\mathbf{C}_k$ and summed to form the 4×3 coefficient matrix $\mathbf{C}_A$.

To compute the depth function $d = \frac{z}{w}$, we perform barycentric interpolation of the z- and w-components of the clip space vertex positions, which are now linear functions of t, e.g., $z(t) = q_z + td_z$ and $w(t) = q_w + td_w$.

Let us consider the depth function, $d(x,y,t)$:

$$d(x,y,t) = \frac{z(x,y,t)}{w(x,y,t)} = \frac{\mathbf{t}_2\sum((q_{kz} + td_{kz})\mathbf{C}_k)\mathbf{x}}{\mathbf{t}_2\sum((q_{kw} + td_{kw})\mathbf{C}_k)\mathbf{x}} = \frac{\mathbf{t}_3\mathbf{C}_z\mathbf{x}}{\mathbf{t}_3\mathbf{C}_w\mathbf{x}}, \tag{13}$$

where:

$$\mathbf{C}_z = \sum(q_{k_z}\underbrace{\begin{bmatrix} & \mathbf{C}_k & \\ 0 & 0 & 0 \end{bmatrix}}_{\overline{\mathbf{C}}_k} + d_{k_z}\underbrace{\begin{bmatrix} 0 & 0 & 0 \\ & \mathbf{C}_k & \end{bmatrix}}_{\underline{\mathbf{C}}_k}), \tag{14}$$

and $\mathbf{C}_w$ is defined correspondingly. We now have the depth function in a convenient form, but the number of coefficients needed are no less than directly storing the vertex positions. We will now examine the contents of the coefficient matrices, $\mathbf{C}_z$ and $\mathbf{C}_w$, in order to simplify their expressions.

Using Equation 14 and the definition of $\mathbf{C}_k$, we can express the first and last row of $\mathbf{C}_w$ as:

$$\begin{aligned} \mathbf{C}_{w_0} &= \sum q_{k_w}\mathbf{h}_k = \sum q_{k_w}\mathbf{q}_i \times \mathbf{q}_j = [0, 0, \det(\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2)], \\ \mathbf{C}_{w_3} &= \sum d_{k_w}\mathbf{f}_k = [0, 0, \det(\mathbf{d}_0, \mathbf{d}_1, \mathbf{d}_2)], \end{aligned} \tag{15}$$

where, in the last step, the terms cancel out to zero for the x- and y-components. The two remaining rows can be simplified in a similar fashion:

$$\begin{aligned} \mathbf{C}_{w_1} &= \sum(q_{k_w}\mathbf{g}_k + d_{k_w}\mathbf{h}_k) \\ &= \sum(q_{k_w}(\mathbf{d}_i \times \mathbf{q}_j + \mathbf{q}_i \times \mathbf{d}_j) + d_{k_w}(\mathbf{q}_i \times \mathbf{q}_j)) \\ &= [0, 0, \sum\det(\mathbf{d}_k, \mathbf{q}_i, \mathbf{q}_j)], \\ \mathbf{C}_{w_2} &= \sum(q_{k_w}\mathbf{f}_k + d_{k_w}\mathbf{g}_k) = [0, 0, \sum\det(\mathbf{q}_k, \mathbf{d}_i, \mathbf{d}_j)]. \end{aligned} \tag{16}$$

Using these expressions, we can formulate $\mathbf{t}_3\mathbf{C}_w\mathbf{x}$ as a cubic function in t independent of (x,y):

$$\mathbf{t}_3\mathbf{C}_w\mathbf{x} = \Delta_0 + \Delta_1 t + \Delta_2 t^2 + \Delta_3 t^3, \tag{17}$$

where:

$$\begin{aligned} \Delta_0 &= \det(\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2), \\ \Delta_1 &= \sum\det(\mathbf{d}_k, \mathbf{q}_i, \mathbf{q}_j), \\ \Delta_2 &= \sum\det(\mathbf{q}_k, \mathbf{d}_i, \mathbf{d}_j), \\ \Delta_3 &= \det(\mathbf{d}_0, \mathbf{d}_1, \mathbf{d}_2). \end{aligned}$$

Expressed differently, the denominator $\mathbf{t}_3\mathbf{C}_w\mathbf{x}$ is the backface status for the moving triangle, e.g., $\det(\mathbf{p}_0(t), \mathbf{p}_1(t), \mathbf{p}_2(t))$ [MAM11].

As a result of these simplifications, we reveal that $\mathbf{t}_3\mathbf{C}_w\mathbf{x}$ has no dependency on x and y and is reduced to a cubic polynomial in t, needing only 4 coefficients. Thus, with this analysis, we have shown that the depth function can be represented by 12 (for $\mathbf{C}_z$) +4 (for $\mathbf{C}_w$) = 16 coefficients, which should be compared to the 24 coefficients needed to store all vertex positions. Our new formulation is substantially more compact.

Further optimization If we use a standard projection matrix, according to Equation 4, we can simplify the depth function further. If we return to Equation 14, and insert the constraint from the projection matrix, i.e., $q_z = aq_w + b$ and $d_z = z_{t_1} - z_{t_0} = a(w_{t_1} - w_{t_0}) = ad_w$, we obtain:

$$\begin{aligned} \mathbf{C}_z &= \sum\left(q_{k_z}\overline{\mathbf{C}}_k + d_{k_z}\underline{\mathbf{C}}_k\right) = \sum\left((aq_{k_w} + b)\overline{\mathbf{C}}_k + ad_{k_w}\underline{\mathbf{C}}_k\right) \\ &= a\mathbf{C}_w + b\sum\overline{\mathbf{C}}_k. \end{aligned} \tag{18}$$

We combine this result with Equation 13 to finally arrive at:

$$\begin{aligned} d(x,y,t) &= \frac{\mathbf{t}_3\mathbf{C}_z\mathbf{x}}{\mathbf{t}_3\mathbf{C}_w\mathbf{x}} = \frac{\mathbf{t}_3(a\mathbf{C}_w + b\sum\overline{\mathbf{C}}_k)\mathbf{x}}{\mathbf{t}_3\mathbf{C}_w\mathbf{x}} = a + b\frac{\mathbf{t}_3(\sum\overline{\mathbf{C}}_k)\mathbf{x}}{\mathbf{t}_3\mathbf{C}_w\mathbf{x}} \\ &= a + b\frac{\mathbf{t}_2(\sum\mathbf{C}_k)\mathbf{x}}{\Delta_0 + \Delta_1 t + \Delta_2 t^2 + \Delta_3 t^3}. \end{aligned} \tag{19}$$

As can be seen above, we have reduced the representation of the depth function from 24 scalar values down to 13 (with the assumption that a and b are given by the graphics API). Later, we will show that this significantly improves the compression ratio for depth functions with motion blur.

Equal Motion Vectors Next, we consider an extra optimization for the special case of linear translation along a vector, since this is a common use case in some applications. In the examples below, we assume that a standard projection matrix is used (i.e., Equation 4). The transformed clip space positions, $\mathbf{p}' = [p'_x, p'_y, p'_w]$, of each triangle vertex are: $\mathbf{p}'_k = \mathbf{q}_k + \mathbf{d}$, where $\mathbf{d} = [d_x, d_y, d_w]$ is a vector in clip space (xyw).

With all motion vectors equal for the three vertices of a triangle, we can derive a simplified depth function. Note that the coefficients $\mathbf{f}_k = 0$, and

$$\det(\mathbf{d}_0, \mathbf{d}_1, \mathbf{d}_2) = \det(\mathbf{d}, \mathbf{d}, \mathbf{d}) = 0,$$

$$\det(\mathbf{q}_i, \mathbf{d}_j, \mathbf{d}_k) = \det(\mathbf{q}_i, \mathbf{d}, \mathbf{d}) = 0. \quad (20)$$

Furthermore, it holds that:

$$\sum \mathbf{g}_k = \sum \mathbf{d} \times (\mathbf{q}_j - \mathbf{q}_i) = \mathbf{d} \times \sum (\mathbf{q}_j - \mathbf{q}_i) = 0. \quad (21)$$

The depth function can then be simplified as:

$$d(x,y,t) = a + b\frac{\mathbf{x} \cdot \sum \mathbf{h}_k}{\Delta_0 + \Delta_1 t}. \quad (22)$$

We have reduced the representation of the depth function from 18 scalar values down to 5 (again with the assumption that a and b are given by the graphics API).

5.2. Depth of Field

There are not as many opportunities to simplify the depth function for defocus blur as there are for motion blur. If we simply store all vertex positions, then $4 \times 3 = 12$ coordinate values are needed. If, however, the projection matrix is known, the number is reduced to $3 \times 3 = 9$. We assume that the camera focal distance and lens aspect are known globally. In the following section, we will show how to reduce the storage requirement of the depth function to 8 scalar coefficients for a defocused triangle.

When depth of field is enabled, a clip-space vertex position is sheared in xy as a function of the lens coordinates (u,v). The vertex position is expressed as:

$$\mathbf{p}_k = \mathbf{q}_k + c_k \mathbf{u}', \quad (23)$$

where c_k is the signed clip space circle of confusion radius, $\mathbf{u}' = [u, \xi v, 0]$, and and ξ is a scalar coefficient that adjusts the lens aspect ratio. Note that c_k is unique for each vertex and is typically a function of the depth. We use these vertices to set up the edge equations:

$$\begin{aligned} e_k(x,y,u,v) &= (\mathbf{p}_i(u,v) \times \mathbf{p}_j(u,v)) \cdot \mathbf{x} \\ &= (\mathbf{q}_i \times \mathbf{q}_j + \mathbf{u}' \times (c_i \mathbf{q}_j - c_j \mathbf{q}_i)) \cdot \mathbf{x} \\ &= (\mathbf{h}_k + \mathbf{u}' \times \mathbf{m}_k) \cdot \mathbf{x}, \end{aligned}$$

where we have introduced $\mathbf{m}_k = (c_i \mathbf{q}_j - c_j \mathbf{q}_i)$ and $\mathbf{h}_k = \mathbf{q}_i \times \mathbf{q}_j$ to simplify notation. With $\mathbf{u} = [u, \xi v, 1]$, we can write the edge equation in matrix form as:

$$e_k(x,y,u,v) = \mathbf{u}\mathbf{C}_k\mathbf{x}, \quad (24)$$

where:

$$\mathbf{C}_k = \begin{bmatrix} 0 & -m_{k_w} & m_{k_y} \\ m_{k_w} & 0 & -m_{k_x} \\ h_{k_x} & h_{k_y} & h_{k_w} \end{bmatrix}. \quad (25)$$

Analogous to the motion blur case, we can express the depth function as a rational function in (x,y,u,v) as follows:

$$d(x,y,u,v) = \frac{z(x,y,u,v)}{w(x,y,u,v)} = \frac{\mathbf{u}\mathbf{C}_z\mathbf{x}}{\mathbf{u}\mathbf{C}_w\mathbf{x}}, \quad (26)$$

where $\mathbf{C}_z = \sum q_{k_z} \mathbf{C}_k$ and $\mathbf{C}_w = \sum q_{k_w} \mathbf{C}_k$. By combining the observation that:

$$\sum q_{k_w} m_{k_w} = \sum q_{k_w} (c_i q_{j_w} - c_j q_{i_w}) = 0, \quad (27)$$

and the top row in Equation 15, $\mathbf{C}_w$ is reduced to a single column, similar to the motion blur case. Thus, the denominator can be written as:

$$\mathbf{u}\mathbf{C}_w\mathbf{x} = \mathbf{u} \begin{bmatrix} 0 & 0 & \sum q_{k_w} m_{k_y} \\ 0 & 0 & -\sum q_{k_w} m_{k_x} \\ 0 & 0 & \det(\mathbf{q}_0, \mathbf{q}_1, \mathbf{q}_2) \end{bmatrix} \mathbf{x} = \Delta_u u + \Delta_v v + \Delta_0. \quad (28)$$

This is equal to $\det(\mathbf{p}_0(u,v), \mathbf{p}_1(u,v), \mathbf{p}_2(u,v))$, which is also the backface status for a defocused triangle [MAM11].

If we introduce the restrictions on the projection matrix from Equation 4, then $\mathbf{C}_z$ can be expressed in the following manner:

$$\mathbf{C}_z = \sum q_{k_z} \mathbf{C}_k = \sum ((a q_{k_w} + b)\mathbf{C}_k) = a\mathbf{C}_w + b\sum \mathbf{C}_k. \quad (29)$$

If we further assume that the clip-space circle of confusion radius follows the thin lens model, it can be written as $c_k = \alpha p_{k_w} + \beta$. With this, we see that:

$$\begin{aligned} \sum m_{k_w} &= \sum (c_i q_{j_w} - c_j q_{i_w}) \\ &= \sum ((\alpha q_{i_w} + \beta) q_{j_w} - (\alpha q_{j_w} + \beta) q_{i_w}) \\ &= \alpha \sum (q_{i_w} p_{j_w} - q_{j_w} p_{i_w}) + \beta \sum (q_{j_w} - q_{i_w}) = 0, \end{aligned} \quad (30)$$

and $\sum \mathbf{C}_k$ takes the form:

$$\sum \mathbf{C}_k = \begin{bmatrix} 0 & 0 & \sum m_{k_y} \\ 0 & 0 & -\sum m_{k_x} \\ \sum h_{k_x} & \sum h_{k_y} & \sum h_{k_w} \end{bmatrix}. \quad (31)$$

With this, we have shown that:

$$d(x,y,u,v) = \frac{\mathbf{u}\mathbf{C}_z\mathbf{x}}{\mathbf{u}\mathbf{C}_w\mathbf{x}} = a + b\frac{\sum \mathbf{h}_k \cdot \mathbf{x} + \sum m_{k_y} u - \sum m_{k_x} \xi v}{\Delta_u u + \Delta_v v + \Delta_0}, \quad (32)$$

which can be represented with 8 scalar coefficients (given that a and b are known). Note that the denominator is linear in each variable.

6. Implementation

We have implemented all our algorithms in a software rasterizer augmented with a depth system [HAM06] containing depth codecs (compressors and decompressors), a depth cache, culling data, and a tile table, which will be described in detail below. To reduce the design space, we chose a cache line size of 512 bits, i.e., 64 bytes, which is a reasonable and realistic size for our purposes. The implication of this choice is that a tile, which is stored using $512 \cdot n$ bits, can be compressed down to $512 \cdot m$ bits, where $1 \leq m < n$ in order to reduce bandwidth usage.

For our results, we present numbers for both 24b integer depth as well as for the complementary depth $(1 - z)$ [LJ99] representation for 32b floating-point buffers. The reason is that this technique has been widely adopted as a superior method on the Xbox 360 (though with 24 bits floating point), since it provides a better distribution of the depths. For all

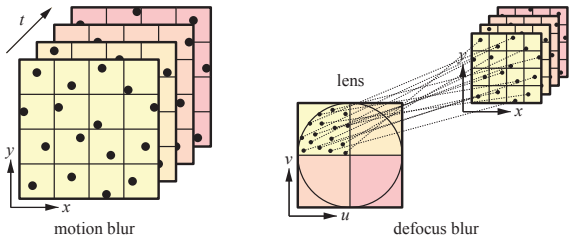


Figure 1: *Left: motion blur for* 4×4 *pixels where there are four samples per pixel (indicated by the four different layers). In total, there are* $4 \times 4 \times 4$ *samples here. If n layers are used we denote such a tile* $4 \times 4 \times n$*. As an example, if each layer is compressed as a separate tile, then we denote these tiles by* $4 \times 4 \times 1$*. Right: we use the same notation for defocus blur, but with a different meaning. Here, the lens has been divided into* 2×2 *smaller lens regions, and as before, there are four samples per pixel (again indicated by the four layers). However, for defocus blur,* $4 \times 4 \times n$ *means that n lens regions are compressed together as a tile.*

our tests, we use a sample depth cache of 64 kB with least-recently used (LRU) replacement strategy.

Even though motion blur is three-dimensional, and defocus blur uses four dimensions, we are using the same tile notation for both these cases in order to simplify the discussion. An explanation of our notation can be found in Figure 1. We perform Z-max culling [GKM93] per $4 \times 4 \times 1$ tiles for 4 spp and $2 \times 2 \times 4$ for 16 spp, where we store z_{max} of the tile using 15 bits. If all of the samples within the tile are cleared, we flag this with one additional bit. If an incoming triangle passed the Z-max test, the per-sample z-test is executed, and the tile's z_{max} is recomputed if any sample pass the per-sample test. For complementary depth z_{min} is used instead.

The tile table, which is accessed through a small cache or stored in an on-chip memory, stores a tile header for each tile. For simplicity, we let the header store eight bits, where one combination indicates that the tile is stored uncompressed, while the remaining combinations are used to indicate different compression modes. In Section 7, we describe which tile sizes have been used for the different algorithms. Using a 32kB cache, we have seen that the total memory bandwidth usage for culling and tile headers is about 10% of the total raw depth buffer bandwidth in general, and approximately the same for all algorithms. Note that culling is essential to performance and is orthogonal to depth buffer compression. Therefore, we chose to exclude those numbers from our measurements and instead just focus on the depth data bandwidth.

Our implementation of the generalized plane encoder, denoted GPE, is straightforward. For motion blur, the rasterizer forwards information about the type of motion applied to each triangle. The three different types of motion that we support are static (no motion), only translation, and arbitrary linear per-vertex motion. In each case, the encoder receives

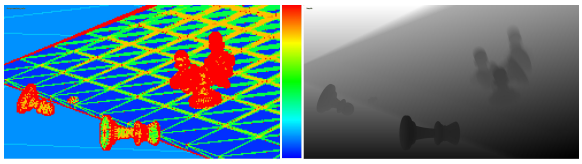


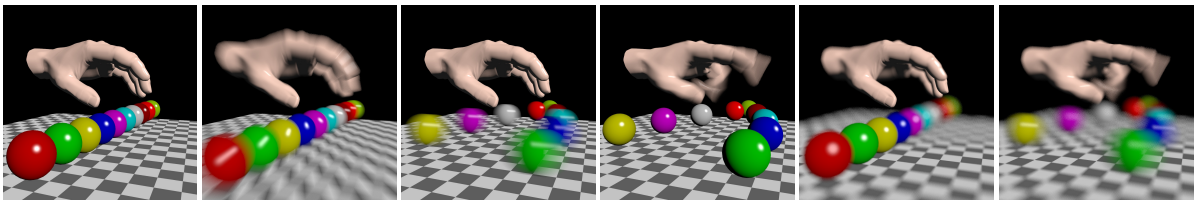
Figure 3: *Left: a false-color visualization of the different* GPE *modes for the Chess scene. Pure red represents uncompressed regions, and blue the modes with the highest compression ratio (16:1). Right: the stochastic depth buffer for the chess scene.*

a set of coefficients, representing the depth function for the current triangle. In addition, the rasterizer forwards a coverage mask, which indicates which samples are inside the triangle. The depth is evaluated for these samples, and depth testing is performed. A depth function of a previously drawn triangle is removed if all of its sample indices are covered by the incoming triangle's coverage mask. The depth of field encoder works in exactly the same way, except that there are no special types for defocus blur that are forwarded.

As shown in the four leftmost images in Figure 2, the number of coefficients needed per triangle is a function of the motion type and varies per triangle from 3 (static), 5 (translation) to 13 (per-vertex motion). Recall that these reductions in the number of coefficients were derived in Section 5. A compressed tile may include x static plane equations, y depth functions for translated triangles, and z depth functions with arbitrary per-vertex motion. The total storage for the depth function coefficients is then $3x+5y+13z$ floats. Additionally, for each sample, we need to indicate which depth function to use (or if the sample is cleared), which is stored with $\lceil \log_2(x+y+z+1) \rceil$ bits per sample. We work on $16 \times 16 \times 1$ tiles for 4 spp and $8 \times 8 \times 4$ tiles for 16 spp, or 16 cache lines, which can be compressed down to one cache line in the best case. Total storage for motion blur is then $(3x+5y+13z) \times 32 + \lceil \log_2(x+y+z+1) \rceil \times 256$. If this sum is less than 16 cache lines, we can reduce the depth bandwidth usage. To simplify the exposition in this paper, we allow only compression to 1, 2, 3, 4, and 8 cache lines, and only use one set of functions for each tile. It is trivial to extend the compressor to the intermediate compression rates as well. Figure 3 shows an example on the usage of the modes in one of our test scenes.

For defocus blur, the expression is simplified, as the depth function is always stored with 8 coefficients per triangle (see Figure 2). If the number of depth functions in a mode is denoted n, the number of bits per mode is given by $8 \times n \times 32 + 256 \lceil \log_2(n+1) \rceil$. The set of modes we have used in this paper is listed in Appendix B.

Note that all coefficients are stored using 32b float, regardless if the depth buffer is 24 or 32 bits. While this precision will not produce the same result as interpolating the depth directly from the vertices, we also would like to note that there is currently no strict rules for how depth should



Static (3) Dolly (5) Translation (5) Per-vertex motion (13) DOF (8) Per-vertex+DOF (18)

Figure 2: *Different configurations of motion blur and depth of field on a simple scene. Below each image, the number of coefficients needed to store the generalized depth function is shown.*

be interpolated in a stochastic rasterizer. If we use the same (compressed) representation in the codec and for depth interpolation, we are guaranteed that the compressed data is lossless. Using the same representation for compression and interpolation makes the algorithms consistent, which is perhaps the most important property. However there is still a question if the interpolation is stable and accurate enough to represent the depth range of the scene. Unfortunately we have not done any formal analysis, and have to defer that to future work.

In absence of any compression/decompression units, it makes more sense to use a tile size that fits into a single cache line [HANAM13]. Therefore, we allow keeping raw data in cache line sized tiles if the compression unit was unable to compress data beyond the raw storage requirement. However, for compressed tiles, we only allow memory transactions from and to the cache of the entire tile. Our baseline, denoted RAW, simply uses uncompressed data on cache line size tiles, which is a much more efficient baseline than previously used [AHAM11, HANAM13] (where the RAW represents uncompressed on the same tile size as the compressed tiles). Since the baseline is more efficient, it means our results are even more significant compared to previous work.

7. Results

In this section, we first describe the set of codecs (compressor and decompressor) we use in the comparison study and then report results on a set of representative test scenes.

Codecs We denote the uncompressed method as RAW below. Note that the RAW mode include Z-max culling and a clear bit per tile, as described in Section 6. An uncompressed $4 \times 4 \times 1$ (4 spp) or $2 \times 2 \times 4$ (16 spp) tile occupies one cache line, i.e., $16 \times 32 = 512$ bits. Our method is denoted GPE, and a detailed implementation description can be found in Section 6.

We compare against a depth offset (DO) codec, working on $8 \times 8 \times 1$ tiles for 4 spp, and $4 \times 4 \times 4$ tiles for 16 spp, where the min and max values are stored in full precision and a per-sample bit indicates if the sample should be delta-encoded w.r.t. the min or the max value. We use three different allocations of the delta bits per sample: 6, 14, and 22. With these layouts, we can compress the four cache lines of depth samples down to one (4:1), two (4:2), and three (4:3) cache lines, respectively. The two bits needed to select one of the three modes or if the sample is cleared are stored in the tile header.

By including time, t, in the predictor functions for a plane encoder, better compression ratios could be achieved for motion blur rasterization [AHAM11]. This technique analyzes all samples in the tile and fits a low-order polynomial surface to the depth samples and encode each sample with an offset from this surface. We include this encoder (denoted AHAM11) in our results and refer to the cited paper for a detailed description of this compression format. We use the same tile sizes as for the DO compressor. Note that unlike GPE, the AHAM11 encoder does not rely on coefficients from the rasterizer, but works directly on sample data. The drawback, however, is that the derivation of the approximate surface and subsequent delta computations are significantly more expensive than directly storing the exact generalized depth function. AHAM11 cannot handle defocus blur.

In a *post-cache* codec the compressor/decompressor is located between the cache and the higher memory levels. In a *pre-cache* codec the compressor/decompressor is located between the Z-unit and the cache, which means that data can be stored in compressed format in the cache. Note that AHAM11 is a post-cache codec, while DO is a pre-cache codec[‡]. GPE is a pre-cache codec as well, similar to plane encoding for static triangles. For a more detailed overview of pre- vs post-cache codecs, we refer to the paper by Hasselgren et al. [HANAM13].

Test Scenes The **Chess** scene contains a mix of highly tessellated geometry (chess pieces) and a coarse base plane. All objects are animated with rigid body motion. The DOF chess scene (32k triangles) has more pieces than the motion blur chess scene (26k triangles). The **Airship** scene (157k triangles) is taken from the Unigine Heaven 2 DX11 demo (on normal tessellation setting), and has been enhanced with a moving camera. This is the only scene tested which uses the

[‡] DO can be either pre- or post-cache codec, but we use pre-cache since it gives better results [HANAM13].

depth functions optimized for camera translation. **Dragon** (162k triangles) shows an animated dragon with skinning and a rotating camera. **Sponza** is the CryTek Sponza scene with a camera rotating around the view vector. The scene has 103k triangles for motion blur and 99k triangles for DOF. Finally, the **Hand** scene (15k triangles) is a key-frame animation of a hand with complex motion. All triangle counts are reported after view frustum culling. Furthermore, in **Airship** and **Dragon**, the triangle counts are after backface culling. All scenes are rendered at 1080p.

The results for motion blur can be seen in Table 1, where the resulting bandwidth for each algorithm is given relative to the RAW baseline. While the numbers reveal substantial savings with our algorithm, it is also interesting to treat the previously best algorithm (which is AHAM11 for most scenes) as the baseline, and see what the improvement is compared to that algorithm. For example, for a complementary depth floating point buffer (F32) at four samples per pixel (spp), we see that the relative bandwidth on the Chess scene is $37/64 \approx 58\%$, which is a large improvement. For Airship, this number is 64%, while it is about 77-80% for Dragon and Sponza. The Hand scene is extremely difficult since it only has per-vertex motion (most expensive) on a densely tessellated mesh, and GPE is unable to compress it further. For 16 spp F32, the corresponding numbers are (from Chess to Hand): 64%, 60%, 80%, 68%, and 89%.

For defocus blur, the results can be found in Table 2. The results are even more encouraging. The relative bandwidth, computed as described above for motion blur, compared to the best algorithm (DO) is (from Chess to Hand): 44%, 74%, 70%, 49%, and 88% for 4 spp. For 16 spp, the corresponding numbers are: 35%, 67%, 66%, 34%, and 67%.

Encoder Complexity Analysis Here, we attempt to do a rough comparison of the complexity of the encoder of GPE vs AHAM11, where we assume that there are n samples per tile. AHAM11 starts by finding min and max of the depths, which results in $\approx 2n$ operations. Each depth value is then binned (n ops) and the largest gap in the bins is found, which splits the samples into two layers. The last step is excluded in our complexity estimate, since it is hard to estimate. For the whole set of samples, and for each of the two layers, a bounding box in x and y is found ($4n$ ops), and the box is split into 2×2 regions, or $2\times 2\times 2$ regions (depending on which mode is used). In each region, the min and the max depth is found ($\approx n$ ops). For each of the two layers and the whole set of samples, the three modes in AHAM11 uses Cramer's rule to compute the predictor function. We estimate this to about 25 FMA (fused multiply-add) operations. The residuals are found by evaluating the predictor and computing the difference. For the three modes, the predictor evaluation costs $4n$, $4n$, and $9n$ ops respectively (including residual computation). Since each sample belongs to the whole set, as well as to one of the two layers, the steps after binning are performed twice per sample. An under-conservative estimation of AHAM11 is then $n(2+1)+2n(4+1+4+4+9) = 47n$ ops plus $9\cdot 25 = 225$ ops for Cramer's rule, i..e, a total of $47n+225$ ops. GPE computes the coefficients, which for the most expensive case (per-vertex motion) costs about 130 FMA ops, and then updates the selection masks, which we estimate to be n operations. Since $47n+225 \gg n+130$ (for $n = 8\times 8$ samples), we conclude that our encoder is more efficient. The per sample cost for reconstructing a depth value is 5 to 13 operations for GPE and 4 to 9 for AHAM11, depending on which depth function or predictor is used. Furthermore, if the stochastic rasterizer performs a backface test, most of the computations needed for the depth function coefficients can be shared. In that scenario, we estimate the constant factor for GPE to be only 20 ops.

8. Conclusions and Future Work

We have presented a generalized plane encoding (GPE) method, where we optimized the depth function representation significantly for motion blur and depth of field separately. GPE provides substantial depth buffer bandwidth savings compared to all previous methods. Our research can have high impact, since we believe that it gets us a bit closer to having a fixed-function stochastic rasterizer in a graphics processor with depth buffer compression.

At this point, we have not been able to provide any positive results for the combination of motion blur and DOF. In future work, we would like to use the theory developed in Appendix A to design more efficient predictors. Although we concluded that the generalized depth function for the case of simultaneous motion blur and depth of field is too expensive in practice, we could analyze the size of each coefficient for a large collection of scenes, and obtain a lower order approximation. As a final step, the residuals would be encoded. Yet another avenue for future research is to explore the depth function for motion blur with non-linear vertex paths.

Acknowledgements The authors thank Aaron Coday, Tom Piazza, Charles Lingle, and Aaron Lefohn for supporting this research. The Advanced Rendering Technology team at Intel provided valuable feedback. Thanks to Denis Shergin from Unigine for letting us use images from Heaven 2.0. Tomas Akenine-Möller is a Royal Swedish Academy of Sciences Research Fellow supported by a grant from the Knut and Alice Wallenberg Foundation.

References

[AHAM11] ANDERSSON M., HASSELGREN J., AKENINE-MÖLLER T.: Depth Buffer Compression for Stochastic Motion Blur Rasterization. In *High Performance Graphics* (2011), pp. 127–134. 2, 7

[AMMH07] AKENINE-MÖLLER T., MUNKBERG J., HASSELGREN J.: Stochastic Rasterization using Time-Continuous Triangles. In *Graphics Hardware* (2007), pp. 7–16. 2, 3

[GDAM10] GRIBEL C. J., DOGGETT M., AKENINE-MÖLLER T.: Analytical Motion Blur Rasterization with Compression. In *High-Performance Graphics* (2010), pp. 163–172. 2, 3

		Chess		Airship		Dragon		Sponza		Hand	
MB		4 spp	16 spp	4 spp	16 spp	4 spp	16 spp	4 spp	16 spp	4 spp	16 spp
RAW (MB)		59.7	256	84.9	355	80.4	366	155	721	36.1	152
DO	F32	84%	75%	100%	95%	92%	89%	93%	88%	100%	98%
	U24	54%	56%	68%	64%	59%	56%	62%	58%	99%	95%
AHAM11	F32	64%	58%	96%	84%	87%	83%	87%	81%	102%	99%
	U24	52%	44%	73%	62%	62%	60%	65%	59%	97%	95%
GPE		37%	37%	62%	50%	70%	66%	67%	55%	100%	88%

Table 1: *Motion blur depth buffer memory bandwidth for both 4 samples per pixel (spp) and 16 spp compared to the baseline,* RAW. *For the comparison methods, we show results for both 32b float, stored as* $1-$depth *(*F32*) and 24b unsigned int (*U24*) depth buffers. By design, the* GPE *scores are identical for 32b float and 24b unsigned int buffers. We disabled the 4:3 compression mode for* DO *float in the hand scene, because otherwise the bandwidth usage rose to* 107% *and* 103%. DO *did, however, benefit from the 4:3 mode in all of our other test scenes. As can be seen, our method (*GPE*) provides substantial bandwidth savings compared to the previous methods in most cases.*

		Chess		Airship		Dragon		Sponza		Hand	
DOF		4 spp	16 spp	4 spp	16 spp	4 spp	16 spp	4 spp	16 spp	4 spp	16 spp
RAW (MB)		102	460	118	413	85.1	413	116	486	40.1	171
DO	F32	88%	86%	100%	99%	93%	92%	86%	82%	99%	95%
	U24	64%	62%	73%	72%	62%	58%	59%	55%	91%	81%
GPE		39%	30%	74%	66%	65%	61%	42%	28%	87%	64%

Table 2: *Depth of field results. Notice that* AHAM11 *is not included here, since it cannot handle defocus blur. Similar to motion blur, we disabled the 4:3 mode for* DO *for 4 spp* F32 *for the hand scene, because otherwise it used* 101% *of the* RAW *bandwidth. Note again that our method (*GPE*) provides significant savings in most test cases. For the comparison method, we show results for both 32b float, stored as* $1-$depth *(*F32*) and 24b unsigned int (*U24*) depth buffers. By design, the* GPE *scores are identical for 32b float and 24b unsigned int buffers.*

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical Z-Buffer Visibility. In *Proceedings of SIGGRAPH 1993* (1993), pp. 231–238. 6

[HAM06] HASSELGREN J., AKENINE-MÖLLER T.: Efficient Depth Buffer Compression. In *Graphics Hardware* (2006), pp. 103–110. 1, 2, 5

[HANAM13] HASSELGREN J., ANDERSSON M., NILSSON J., AKENINE-MÖLLER T.: A Compressed Depth Cache. *to appear in Journal of Computer Graphics Techniques* (2013). 2, 7

[LJ99] LAPIDOUS E., JIAO G.: Optimal Depth Buffer for Low-Cost Graphics Hardware. In *Graphics Hardware* (1999), pp. 67–73. 5

[MAM11] MUNKBERG J., AKENINE-MÖLLER T.: Backface Culling for Motion Blur and Depth of Field. *journal of graphics, gpu, and game tools, 15*, 2 (2011), 123–139. 4, 5

[MAM12] MUNKBERG J., AKENINE-MÖLLER T.: Hyperplane Culling for Stochastic Rasterization. In *Eurographics – Short Papers* (2012), pp. 105–108. 10

[Mor00] MOREIN S.: ATI Radeon HyperZ Technology. In *Workshop on Graphics Hardware, Hot3D Proceedings* (August 2000), ACM Press. 1

[MWM02] MCCOOL M. D., WALES C., MOULE K.: Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization. In *Graphics Hardware* (2002), pp. 65–72. 2

[PLS12] POOL J., LASTRA A., SINGH M.: Lossless Compression of Variable-Precision Floating-Point Buffers on GPUs. In *Symposium on Interactive 3D Graphics and Games* (2012), pp. 47–54. 2

[RHAM07] RASMUSSON J., HASSELGREN J., AKENINE-MÖLLER T.: Exact and Error-bounded Approximate Color Buffer Compression and Decompression. In *Graphics Hardware* (2007), pp. 41–48. 2

[RSAM08] RASMUSSON J., STRÖM J., AKENINE-MÖLLER T.: Error-Bounded Lossy Compression of Floating-Point Color Buffers using Quadtree Decomposition. *The Visual Computer, 26*, 1 (January 2008), 17–30. 2

[SWR*08] STRÖM J., WENNERSTEN P., RASMUSSON J., HASSELGREN J., MUNKBERG J., CLARBERG P., AKENINE-MÖLLER T.: Floating-Point Buffer Compression in a Unified Codec Architecture. In *Graphics Hardware* (2008), pp. 96–101. 2

Appendix A - Motion Blur + Depth of Field

Following the derivation for motion blur and depth of field, we want to create general depth functions for the case of triangles undergoing simultaneous motion blur and depth of field. We first consider the 5D edge equation [MAM12]:

$$e_k(x,y,u,v,t) = (\mathbf{n}_k(t) + \mathbf{u}' \times \mathbf{m}_k(t)) \cdot \mathbf{x}, \quad (33)$$

where $\mathbf{u}' = (u, \xi v, 0)$, $\mathbf{n}_k(t) = \mathbf{p}_i(t) \times \mathbf{p}_j(t)$, and $\mathbf{m}_k(t) = c_i(t)\mathbf{p}_j(t) - c_j(t)\mathbf{p}_i(t)$. The interpolation formula becomes:

$$A(x,y,u,v,t) = \frac{\sum A_k e_k(x,y,u,v,t)}{\sum e_k(x,y,u,v,t)}. \quad (34)$$

Our first goal is to derive a compact formulation of the denominator in Equation 6:

$$\begin{aligned} &\sum p_{k_w}(t) e_k(x,y,u,v,t) \\ &= \sum p_{k_w}(t)\mathbf{n}_k(t) \cdot \mathbf{x} + \sum (p_{k_w}(t)\mathbf{u}' \times \mathbf{m}_k(t)) \cdot \mathbf{x}. \end{aligned} \quad (35)$$

From Equation 17, we have: $\sum p_{k_w}(t)\mathbf{n}_k(t) \cdot \mathbf{x} = \sum_0^3 \Delta_i t^i$. Similarly, by generalizing Equation 27, we obtain:

$$\sum p_{k_w}(t) m_{k_w}(t) = 0, \quad (36)$$

which can be used to simplify the term below:

$$\begin{aligned} &\sum (p_{k_w}(t)\mathbf{u}' \times \mathbf{m}_k(t)) \cdot \mathbf{x} \\ &= \left(0, 0, u\sum p_{k_w}(t) m_{k_y}(t) - \xi v \sum p_{k_w}(t) m_{k_x}(t)\right) \cdot \mathbf{x} \\ &= u\sum_0^3 \gamma_i t^i + \xi v \sum_0^3 \delta_i t^i. \end{aligned} \quad (37)$$

Simplification for the thin lens model If we assume that the clip space circle of confusion radius follows the thin lens model, it can be written as $c_k(t) = \alpha p_{w_k}(t) + \beta$. We use the equality:

$$\sum (p_{k_w}(t)\mathbf{u}' \times \mathbf{m}_k(t)) \cdot \mathbf{x} = \mathbf{u}' \cdot \sum c_k(t)\mathbf{p}_i(t) \times \mathbf{p}_j(t), \quad (38)$$

and see that:

$$\begin{aligned} &\mathbf{u}' \cdot \sum c_k(t)\mathbf{p}_i(t) \times \mathbf{p}_j(t) \\ &= \mathbf{u}' \cdot \sum ((\alpha p_{k_w}(t) + \beta)\mathbf{p}_i(t) \times \mathbf{p}_j(t)) \\ &= \beta \mathbf{u}' \cdot \sum (\mathbf{p}_i(t) \times \mathbf{p}_j(t)) = u\sum_0^2 \gamma_i t^i + \xi v \sum_0^2 \delta_i t^i. \end{aligned} \quad (39)$$

We have shown that the denominator can be expressed as:

$$\sum p_{k_w}(t) e_k(x,y,u,v,t) = u\sum_0^2 \gamma_i t^i + \xi\, v \sum_0^2 \delta_i t^i + \sum_0^3 \Delta_i t^i, \quad (40)$$

which can be represented by 10 coefficients. The numerator:

$$\sum e(x,y,u,v,t) = (\sum \mathbf{n}_k(t) + \mathbf{u}' \times \sum \mathbf{m}_k(t)) \cdot \mathbf{x}, \quad (41)$$

can be represented by 18 coefficients, but again, if we assume that the clip space circle of confusion radius follows the thin lens model, we can generalize Equation 31 and see that $\sum m_{k_w}(t) = 0$. Then we obtain $(\mathbf{u}' \times \sum \mathbf{m}_k(t)) \cdot \mathbf{x} = u\sum m_{k_y}(t) - \xi v \sum m_{k_x}(t) = u\sum_0^2 \lambda_i t^i + \xi v \sum_0^2 \kappa_i t^i$.

Mode	x	y	z	c	cache lines
0	29	0	0	1	8
1	0	17	0	1	8
2	0	0	8	0	8
3	0	0	7	1	8
4	10	0	0	1	4
5	0	8	0	0	4
6	0	7	0	1	4
7	0	0	3	1	4
8	8	0	0	0	3
9	7	0	0	1	3
10	0	4	0	1	3
11	0	0	2	1	3
12	4	0	0	0	2
13	0	3	0	1	2
14	0	0	1	1	2
15	2	0	0	0	1
16	0	1	0	1	1
17	0	0	1	0	1

Table 3: *GPE compression modes for motion blur.*

Mode	number of planes	c	cache lines
0	12	1	8
1	5	1	4
2	4	0	3
3	3	1	3
4	2	1	2
5	1	1	1

Table 4: *GPE compression modes for defocus blur.*

Thus, we can represent the depth function, $d = z/w$, with 25 coefficients. Note that simply storing the vertices, $[x,y,w]$, would require $3 \times 3 \times 2 = 18$ values, which is a more compact representation. We conclude that for the combination of motion and defocus blur, the raw vertex representation is a better alternative in term of storage. Our derivation was still included in order to help others avoid going down this trail of simplifying the equations.

Appendix B - Compression modes for GPE

The total storage cost of a compressed block is: $(3x + 5y + 13z) \times 32$ bits for the depth function coefficients plus $\lceil \log_2(x+y+z+1) \rceil \times 256$ bits to indicate which depth function to use for each of the 256 samples (or if the sample is cleared). As an additional optimization, if no samples are cleared, we skip the clear bit in some modes. If a clear bit is present in the mode, this is indicated as $c = 1$ in Table 3. Similarly, we show the modes for defocus blur in Table 4. We empirically found a reasonable subset of the large search space of possible predictor combinations that worked well in our test scenes.