Depth Buffer Compression for Stochastic Motion Blur Rasterization

Magnus Andersson[1,2]   Jon Hasselgren[1]   Tomas Akenine-Möller[1,2]

[1]Intel Corporation   [2]Lund University

Abstract

Previous depth buffer compression schemes are tuned for compressing depths values generated when rasterizing static triangles. They provide generous bandwidth usage savings, and are of great importance to graphics processors. However, stochastic rasterization for motion blur and depth of field is becoming a reality even for real-time graphics, and previous depth buffer compression algorithms fail to compress such buffers due to the irregularity of the positions and depths of the rendered samples. Therefore, we present a new algorithm that targets compression of scenes rendered with stochastic motion blur rasterization. If possible, our algorithm fits a single time-dependent predictor function for all the samples in a tile. However, sometimes the depths are localized in more than one layer, and we therefore apply a clustering algorithm to split the tile of samples into two layers. One time-dependent predictor function is then created per layer. The residuals between the predictor and the actual depths are then stored as delta corrections. For scenes with moderate motion, our algorithm can compress down to 65% compared to 75% for the previously best algorithm for stochastic buffers.

CR Categories: I.3.1 [Computer Graphics]: Hardware architecture—Graphics processors I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: stochastic rasterization, depth buffer compression, motion blur.

1 Introduction

In general, graphics processors are dependent on a number of techniques to reduce memory bandwidth usage. A memory access may cost several orders of magnitudes more in terms of power consumption than an arithmetic operation [Dally 2009], and the gap between compute power and the available bandwidth continues to grow. Hence, it is well worth the silicon to add fixed-function units for those techniques.

Over the past few years, a lot of effort [Akenine-Möller et al. 2007; Toth and Linder 2008; Fatahalian et al. 2009; McGuire et al. 2010; Brunhaver et al. 2010] has been put into getting stochastic rasterization [Cook et al. 1987] of motion blur and depth of field closer to interactive or even real-time rendering. The characteristics of stochastic rasterization is likely to influence some of the techniques for reducing usage of memory bandwidth, especially the ones based on compression.

Depth buffer compression is one very important technique [Morein 2000] to reduce memory bandwidth usage. We review the known
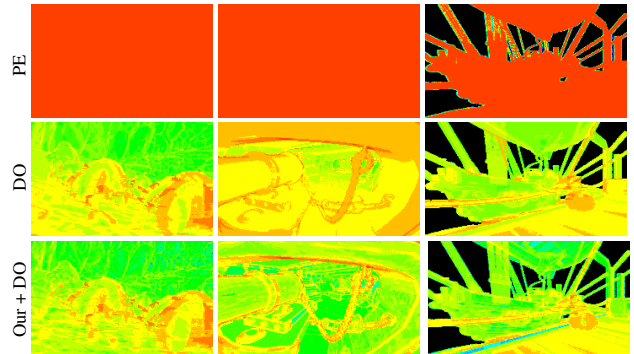

Figure 1: *False color visualizations of depth buffer compression ratios of some scenes, where our algorithm outperforms the competition for compressing stochastic motion blur depth buffers. Plane encoding (PE) is one of the best algorithms for compressing static scenes, and depth offset (DO) compression is the best existing scheme for handling "noisy" buffers. As can be seen, plane encoding breaks down completely due to that plane equations turn into higher order rational polynomials when motion is introduced. Depth offset compression is substantially better, but the combination of our novel time-dependent compression algorithm with DO is even better. The color scale is, from low compression ratio (good) to high compression ratio (bad), blue, cyan, green, yellow, and red (which represents uncompressed).*

algorithms [Morein 2000; Hasselgren and Akenine-Möller 2006; Lloyd et al. 2007; Ström et al. 2008] in the last part of Section 2. Note that lossy compression of other depth buffer representations can also be done [Gribel et al. 2010; Salvi et al. 2010], but these do not solve the problem of compression of stochastically generated buffers.

Most existing algorithms depend of the fact that depth, z, is linear over a triangle, and this is exploited to construct inexpensive compression and decompression algorithms. One of the state-of-the-art algorithms, called *plane encoding* (described by Hasselgren and Akenine-Möller [2006]), is particularly good at exploiting this. Briefly, the rasterizer feeds exact plane equations to the compressor, and hence do not need any residual terms. However, for motion blur, where each vertex may move according to a linear function, the depth function at a sample is a rational cubic function [Gribel et al. 2010]. This fact makes it substantially harder to predict depth over an entire tile using a simple predictor function. As a consequence, the standard depth buffer compression techniques, especially the ones exploiting exact plane equations, will in many cases fail to compress such "noisy" buffers.

We present a novel technique for compression of stochastic depth buffers generated with motion blur. Our technique is able to compress a substantial amount of blocks of pixels, where previous techniques break down. This can be seen in Figure 1. One of the best algorithms, *plane encoding*, for static scenes breaks down completely. The best existing algorithm for compressing noisy depth buffers is called *depth offset* compression, and as can be seen in the same figure, that algorithm has decent performance for motion blur renderings. We believe our technique could become important in order to bring stochastic rasterization into fixed-function units in graphics processors.
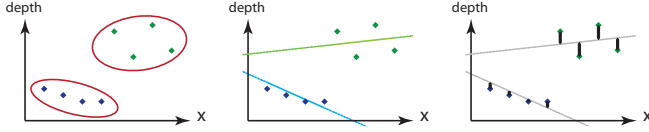
Figure 2: *Illustration of the three steps of a depth buffer compression algorithm. These are, from left to right,* 1) *clustering,* 2) *predictor function generation, and* 3) *residual encoding.*

2 Compression Framework

Here, we present a very simple general framework (Section 2.1) that can be used to describe all existing depth buffer compression schemes that we know of, which is done in Section 2.2.

2.1 Framework

Let us start with some assumptions. A block of $w \times h$ pixels, sometimes called a *tile*, is processed independently, and we assume that each pixel has n samples. The i:th sample is denoted by $\mathbf{s}^i = (s_x^i, s_y^i, s_t^i, s_z^i)$, where the first two components are the x- and y-coordinates of the sample inside the tile, and the third component, $s_t^i \in [0, 1]$, is the time of the sample. It is also possible to add more components, for example, (s_u^i, s_v^i), for the lens position for depth of field rendering. Current depth compression schemes do not handle motion blur and depth of field explicitly, and hence do not have the time component nor the lens parameters. Note that all of (s_x^i, s_y^i, s_t^i) are fixed for a particular sample. It is only the depths that results from the rasterization process, and as a consequence, it is only the depth values, s_z^i, that needs to be compressed. Our notation for depth here is $s_z^i = z_c / w_c$, where z_c and w_c are the z- and w-components of a sample in clip-space, as usual. In general, a compression algorithm may attempt to exploit the fixed components for better compression.

From studying the sparse set of previous work on depth buffer compression [Morein 2000; Hasselgren and Akenine-Möller 2006; Lloyd et al. 2007; Ström et al. 2008], we realized that all known schemes share three common steps, namely:

1. clustering,
2. predictor function generation, and
3. residual encoding.

These three steps are illustrated in Figure 2. It should be noted, though, that an algorithm may choose not to have one or two of the steps above. A high level description of each of the steps follows.

Clustering is needed when there are, for example, a set of samples in a tile that belongs to a background layer, and the rest of the samples in the tile belongs to a foreground layer. In these cases, it is very hard to compress all depths in the tile using the same predictor function. The clustering step therefore attempts to separate the samples of a tile into two or several *layers*, where the samples in each layer typically should share some characteristics (e.g., lie in a common plane). The goal of splitting the samples into two or more layers is that each layer should ideally become simpler to compress compared to compressing all samples as a single layer. For a tile with only foreground samples though or when only one triangle covers an entire tile, clustering may not be needed. In general, a bitmask or several bitmasks are needed to indicate which layer a sample belongs to.

As the next step, each layer generates its own *predictor function*. The goal here is to use the depth samples and possibly their fixed (x, y, t)-coordinates to create a predictor function, $z(x, y, t)$,

whose task is to attempt to predict the depth at each sample using an inexpensive (in terms of storage, generation, and evalation) function. For example, assume that a rectangle with small per-pixel displacements has been rendered to a tile. As a predictor function, one may use the plane of the rectangle, since it probably is a good guess on where the displaced depths will be. This guess will not be 100% correct, and so it is up to the next step to correct this.

Residual encoding must make sure that the exact depths, s_z^i, can be reconstructed during decompression of the tile, since a common requirement by graphics APIs is that the depth buffer is non-lossy. The residual, which is the difference between the predictor function, $z(x, y, t)$, and a sample's depths, is computed as:

$$\delta_i = z(x, y, t) - s_z^i. \tag{1}$$

Given a good predictor function, the residuals, δ_i, between the depth of the samples and the predictor function should be small. As a consequence, the deltas can be encoded using few bits. Good compression ratios can be achieved if there are a small number of layers, storage (in bits) for predictor function is small, and if the deltas can be encoded using few bits as well. Another success factor of a compression scheme is that the algorithm should succeed in compressing many tiles during rendering.

2.2 Depth Compression Algorithms

In this subsection, we will briefly describe the existing depth buffer compression algorithms in terms of our introduced framework. We will use the same names of the algorithms as introduced in the survey of depth buffer compression algorithms [Hasselgren and Akenine-Möller 2006], and we will also describe Lloyd et al's algorithm [2007].

The *depth offset* algorithm uses the $z_{\min}$ and $z_{\max}$ of the tile to cluster the samples into two layers; one being closer to $z_{\min}$ and the other with samples being closer to $z_{\max}$. The predictor functions are as simple as they could be — for the closer layer, the differences between the sample's depth and $z_{\min}$ is encoded, and vice versa. In the *differential differential pulse code modulation* (DDPCM) algorithm, the idea is to compute first-order and second-order differentials, which essentially generate a plane equation as the predictor function. The differences between this plane equation and the actual depth values are encoded using two bits per sample. An extension is discussed, where a search for two different layers is performed, and each layer is encoded using a plane equation. Hence, this is a clustering step. *Anchor encoding* is very similar to DDPCM. As a predictor function, a plane equation is created from an anchor position in the tile, and two delta depth values stored as part of the plane equation. The residuals are encoded using five bits, and so could potentially be more useful for scenes rendered with lots of small triangles. Clustering is not used in this algorithm.

Plane encoding uses information from the rasterizer to create the clustering and exact plane equations. The rasterizer can assist the compression algorithm with bitmasks indicating which samples are covered by the triangle being rasterized, which means that clustering is done implicitly. In addition, the rasterizer can also provide full accuracy plane equations, meaning that there will be no residuals to encode, and so this is an example where the last step is missing. If the rasterizer is disconnected from the compression unit, a search algorithm for clustering into two layers can be used [Hasselgren and Akenine-Möller 2006]. Each layer is then encoded using a plane equation with only a single bit as a correction factor, which gives a bit better performance.

Ström et al. [2008] presented the first public algorithm for compressing floating-point depth buffers. The floating-point numbers

were interpreted as integers in order to be able to encode differences. They used small set of previously decompressed depth values to feed a predictor function, and Golomb-Rice for entropy encoding of the residuals.

Lloyd et al. [2007] develop a compression algorithm specifically for logarithmic shadow maps. In this case, the planar triangles become curved, and linearity cannot be exploited. Instead, first-order differentials are first computed, as done in DDPCM, and then anchor encoding is used on the differentials. No clustering is done for this method.

As can be seen, most of these algorithms use some kind of plane equation of the geometry as predictor functions. With motion blur, the depth at each sample for a single triangle is a cubic rational polynomial [Gribel et al. 2010], and so it should be clear that they stand little chance at being successful at compressing scenes with stochastic motion blur (or depth of field). Some of the efficiency of these algorithms also stem from the assumption that the samples are positioned in a regular grid in xy, which is not true for stochastic samples. Depth offset does not use any such assumptions, but on the other hand, that algorithm uses the simplest possible clustering and also the simplest predictor functions. Akenine-Möller et al. [2007] ran some initial tests on using depth offset compression for stochastic buffers, and found that it worked reasonably well, but conjectured that better algorithms should be possible. In the following sections, we will present new ways to approach stochastic depth compression using our framework.

3   New Algorithms

In this section, we describe our new contributions to the field of depth compression. First, we describe a simple clustering technique that can be applied to both stochastic depth buffers and to depth buffers rendered without blur. In Section 3.2, we introduce *time-dependent* predictor functions, and finally, we also describe how residual encoding is done in Section 3.3.

3.1   Clustering

In this subsection, we present a novel clustering technique, which is extremely simple and rather inexpensive to implement in hardware. Usually, two depth layers within a tile are separated by a large gap in depth. To find this separation, a simple approach would be to sort the samples according to their depths, and then find the largest depth difference between two successive samples. However, this is much too expensive for a hardware implementation, which needs to avoid sorting in order to reduce complexity. Instead, we propose a more pragmatic approach, which is not optimal, but tends to give good results. First, we split the depth interval between z_{min} and $\hat{z}_{max}$ for the tile into n bins, where $\hat{z}_{max}$ is the maximum z-value of non-cleared samples. For each bin, we store one bit, which records whether there is at least one sample in the bin. The bits are initiated to zero. Each sample is then classified to a bin based on the sample's depth value, and the corresponding bit set to one. Samples that are cleared may be ignored in this step. When all samples have been processed, each 0 signals a gap in depth of at least $(\hat{z}_{max} - z_{min})/n$. By finding the largest range of consecutive zeroes, a good approximation of the separation of the two depth layers is obtained. This entire process is illustrated in Figure 3. Each of the sample clusters produced by this step is then processed independently as a layer by the predictor function generation step. In addition, the clustering process implicitly generates one or more bitmasks that indicate which layer each sample belongs to. The bitmask(s) will be part of the compressed representation of a tile. If needed, the samples can be clustered into more layers simply by
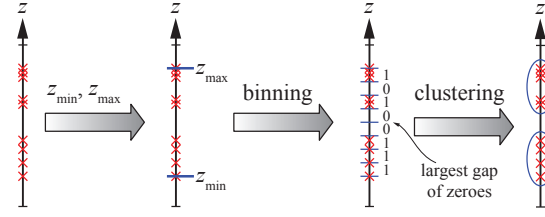

Figure 3: *Illustration of our new clustering technique. From left to right: the depth values are marked as red crosses on the depth axis, and these depth values are then bounded by z_{min} and z_{max}. Then follows binning where, in this case, eight small bins between z_{min} and z_{max} are created, and bins with at least one depth sample are marked with 1, and otherwise marked with 0. Finally, the largest gaps of zeroes is found, and this separates the depths into two layers.*

finding the second and third (and so on) longest ranges of consecutive zeroes.

3.2   Predictor Functions

At this point, we have a bitmask, generated from the previous step, indicating which of the tile's $w \times h \times n$ samples that should be compressed for the current layer. Note that we may have only one layer, in which case all samples are included.

As mentioned earlier, most depth buffer compression schemes rely on that depth, $z = z_c/w_c$, is linear in screen space across a triangle. This means that a planar predictor function, such as the one shown below, is frequently used.

$$z(x, y) = a + bx + cy. \tag{2}$$

However, as soon as the time dimension is included so that motion blur is rendered, this is no longer ideal. We approach the problem of compressing stochastic buffers generated with motion blur by adding the time, t, to the predictor, but also combine x, y and t in different permutations. In general, we can use a predictor function, $z(x, y, t)$ as follows:

$$z(x, y, t) = \sum_{mno} a_{mno} x^m y^n t^o. \tag{3}$$

For a hardware compressor, it is not feasible to try all possible combinations when performing compression, and having too many terms makes it very expensive to compute the predictor function. Based on the equation above, we propose to use three predictor functions, which have not been used in depth compression before. They were chosen due to their simple nature (few coefficients and low degree of the terms). For future work, it may be interesting to attempt to use other predictor functions, with higher degree polynomial terms, as well. Our new modes are listed below:

Mode	Equation
0: $Patch(x, y)$	$z_0(x, y) = a + bx + cy + dxy$
1: $Plane(x, y, t)$	$z_1(x, y, t) = a + bx + cy + dt$
2: $Patch(x, y, t)$	$z_2(x, y, t) = (1 - t)(a_0 + b_0x + c_0y + d_0xy)$ $+ t(a_1 + b_1x + c_1y + d_1xy)$

Our three modes have predictor functions with 4 or 8 unknown coefficients (a, b, etc.). These can be obtained in a number of ways. However, instead of using conventional solutions with a high computational cost, we will instead use an inexpensive, approximate method to do this.

Since each tile contains many samples, it is possible to set up an over-constrained linear system when determining the coefficients

of the predictor functions. For the predictor function, $z_1(x,y,t) = a + bx + cy + dt$ (mode 1 above), this would be done as shown below (where a has been removed, because it is computed in the final stage of the algorithm):

$$\underbrace{\begin{pmatrix} s_x^0 & s_y^0 & s_t^0 \\ s_x^1 & s_y^1 & s_t^1 \\ & \vdots & \\ s_x^{m-1} & s_y^{m-1} & s_t^{m-1} \end{pmatrix}}_{\mathbf{M}} \underbrace{\begin{pmatrix} b \\ c \\ d \end{pmatrix}}_{\mathbf{k}} = \underbrace{\begin{pmatrix} s_z^0 \\ s_z^1 \\ \vdots \\ s_z^{m-1} \end{pmatrix}}_{\mathbf{z}} \Leftrightarrow \mathbf{Mk} = \mathbf{z}, \quad (4)$$

where m is the number of samples in the current layer.

At a first glance, it may seem like a good idea to use least-squares fitting to solve this problem. The unknown is $\mathbf{k}$, and such an over-constrained system is often solved by performing a (costly) multiplication with the transpose of $\mathbf{M}$, which gives a square (and hence, possibly invertible) matrix: $\mathbf{M}^T\mathbf{Mk} = \mathbf{M}^T\mathbf{z}$. The solution is then: $\mathbf{k} = (\mathbf{M}^T\mathbf{M})^{-1}\mathbf{M}^T\mathbf{z}$, which is often called a pseudo-inverse [Golub and Loan 1996], and gives us a solution in the least-squares sense (i.e., minimizing the errors in the 2-norm). Note that for the example in Equation 4, we need to invert a 3×3 matrix to solve for b, c, and d. This can be done with Cramer's rule,

While this would make for a decent estimation, our real goal is to minimize the maximum difference between the samples' depths and the predictor function, since this minimizes number of bits needed for the residual encoding (see Section 3.3). This can be done using minimax fitting [Houle and Toussaint 1988], which is an even more expensive algorithm than least squares fitting. Since both these approaches are too expensive, we propose instead to use a heuristic data reduction technique. We reduce the samples in a layer into a more manageable number of *representative points*, which can then be used to solve a small 3×3 linear system. These points should be selected in a manner such that the resulting prediction function lies as close to the minimax solution as possible.[1]

Data reduction: The following algorithm is used to compute the representative sample points. When time, t, is *not* included (e.g., for mode 0), the first step is to find the bounding box in x and y for all the samples in the layer. The bounding box is then split into 2×2 uniform grid cells. For each cell, we find the two samples with the minimum and maximum depth values. The mid-point of these two samples (in xyz) is then computed. This gives us four representative points, $\mathbf{r}^{ij} = (r_x^{ij}, r_y^{ij}, r_t^{ij}, r_z^{ij})$, with $i, j \in \{0, 1\}$, where i and j are grid cell coordinates in our 2×2 grid. There will be at most four representative points, and these will be used to compute the predictor function inexpensively. Analogously, for modes that takes t into account (mode 1 & 2), we can compute the bounds in t as well, and instead split the bounding box into $2 \times 2 \times 2$ grid cells. This results in at most 8 representative points, $\mathbf{r}^{ijk}$, with $i, j, k \in \{0, 1\}$. An illustration of the data reduction algorithm for the 2×2 case can be found in Figure 4.

Next, we describe how we compute each mode's predictor function from these reduced representative data points.

Common step: All our three modes share a common step when computing their predictor functions. They all need to solve one or two 4×4 systems of linear equations to get the coefficients for the predictor function. To simplify this, we first move the origin to

[1]Some experimental results were obtained by comparing our solution for a 4D plane (mode 1) to Matlab's least squares (backslash operator) and minimax (fminimax) solutions of the over-constrained system. A set of random tiles were retrieved from our test scenes, and the mean error span was calculated using all three methods. We were well within 10% of the minimax error, and on par with the least squares solution.

one of the representative points, and instead compute a later in the residual encoding step (see Section 3.3). This leaves three coefficients left to solve, and three remaining representative points. Any method suitable for solving 3×3 linear systems, such as Cramer's rule, can be used to compute these.

Mode 0: This mode was added mainly for static geometry, i.e., for parts of the rendered frame without motion blur. Therefore, it does not contain the time parameter. However, it will also be used in mode 2, as will be seen later. We propose to use a bilinear patch, which is described by $z_0(x,y) = a + bx + cy + dxy$. The motivation for this mode, compared to using just plane equations (see Section 2), is that the bilinear patch is somewhat more flexible, since it is a second-degree surface, and hence has a higher chance of adapting to smoother changes of the surface.

Mode 1: This mode describes a plane in four dimensions, i.e., $z_1(x,y,t) = a + bx + cy + dt$. This representation is useful for moving geometry, since it contains the dt term. In this case, we can use any four representative points, $\mathbf{r}^{ijk}$, where at least one has $k = 0$ and at least one has $k = 1$ in order to capture time dependence. Although a plane in four dimensions is not enough to capture all possible triangle movements, this approximation works surprisingly well, as we will see in Section 5.

Mode 2: This mode linearly interpolates two bilinear patches (P_0 and P_1) positioned at $t = 0$ and at $t = 1$, to capture a surface moving over time. The resulting equation becomes: $z_2(x,y,t) = P_0 + t(P_1 - P_0)$. To compute this representation, we first perform data reduction to produce $2 \times 2 \times 2$ representative points, $\mathbf{r}^{ijk}$. For $k = 0$, the four representative points, $\mathbf{r}^{ij0}$, $i, j \in \{0, 1\}$, are used to compute the P_0 patch in the same way as done for mode 0. A similar procedure is used to compute P_1. Each patch, P_k, now approximately represents the tile data at the time $t_k = \frac{\max(r_t^{ijk}) + \min(r_t^{ijk})}{2}$. We now have all eight coefficients needed for this mode. In a final step, the two patches are positioned at times $t_0 = 0$ and $t_1 = 1$ through extrapolation, which gives us $z_2(x,y,t)$.

Missing data: Due to clustering and cleared samples, some grid cells in the data reduction step may end up without any samples, which means that representative points, $\mathbf{r}^{ij(k)}$, for such grid cells cannot be computed. To be able to compute the predictor functions, we choose to make reasonable estimations of the missing representative points using the existing representative points. For simplicity, we only fill in missing data over the xy neighbors, and not in t. Hence, for time-dependent modes, we use the same technique twice. If only one representative point is missing, i.e., one grid cell is missing samples, we create a plane from other three points, and evaluate it at the center of the empty grid cell.

If there are only two representative points, e.g., $\mathbf{r}^{00}$ and $\mathbf{r}^{01}$, and two empty grid cells, we create a new point, $\mathbf{r}^{10}$, as shown below:

$$\begin{aligned} \mathbf{e} &= \mathbf{r}^{01} - \mathbf{r}^{00}, \\ \mathbf{r}^{10} &= (r_x^{00} - e_y, r_y^{00} + e_x, r_t^{00}, r_z^{00}), \end{aligned} \quad (5)$$

where the first two components of $\mathbf{r}^{10}$ are created by rotating the difference vector, $\mathbf{e}$, 90 degrees in the xy-plane and adding it to the x and y of $\mathbf{r}^{00}$. The other components are simply copied from $\mathbf{r}^{00}$. This extrudes a plane from the vector from $\mathbf{r}^{00}$ to $\mathbf{r}^{01}$. When this third representative point has been created, we proceed as if only one representative point is missing. Finally, if only one representative point exists, this implies that the layer only contains a single sample. In such a case, only the a-coefficient is needed since it encodes a constant function, which is all we need to reconstruct a single depth value. As a consequence, the representative points are not needed in this case.
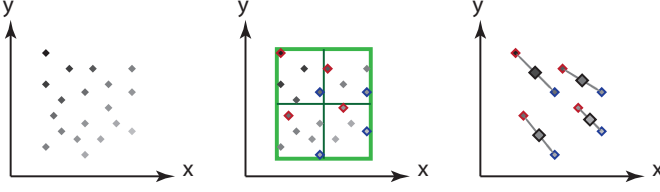
Figure 4: *All our predictor functions uses a similar data reduction step. This figure illustrates how a set of samples are reduced to 2×2 representative points. (**left**) We start with a set of irregular samples. The grayscale of the samples indicate their depths. (**middle**) The bounding box of the samples is found. The box is split in half in x and y. In each resulting sub-region, the samples with the maximum (blue) and minimum (red) depths are found. (**right**) The mean position and depth for each pair of min- and max-samples is used as a new representative point for the sub-region.*

3.3 Residual Encoding

In a final step, we compute correction terms that encode how a specific sample can be recreated from the predictor functions. We need to store two values for every sample. The first is a *layer index*, which associates that sample with a certain layer, as described in Section 3.1. Typically, we use only between one and two layers, so we need at most one bit per sample for this index. If a tile can be compressed using a single layer, we do not have to store these indices.

The second per-sample values to store are the correction terms, δ_i. These are found by looping over all of samples in the layer and computing the difference between the predicted value, $z(x, y, t)$, and the actual depth of the sample, s_z^i. During this phase, we track the required number of bits to store the correction terms, and also compute the a-constant for our predictor functions. The a-constant is set so that we only get unsigned correction terms (i.e., all samples lie above the predictor function).

Our correction terms are used in a slightly new way. For k correction bits per correction term, we reserve the value $2^k - 1$ as a *clear* value, and can hence only use correction terms of up to (and including) $2^k - 2$. However, we get the benefit of being able to signal whether a particular sample is cleared in a very inexpensive way. Otherwise, this is usually done using a particular value in the layer index, which is more costly.

4 Implementation

We use a depth compression architecture with a tiled depth buffer cache and a tile table [Hasselgren and Akenine-Möller 2006]. This has been implemented in a software rasterizer in order to gather statistics about different algorithms and configurations. In order to reduce the dimensionality of the evaluation space, we decided to use 512 bits, that is 64 bytes, as the cache line width in our simulations. This implies that when we compress a tile, the compressed representation is always padded to the next 512-bit alignment regardless of the size of the desired memory transaction. In all of our tests, we use 32-bit fixed-point depth values in the depth buffer.

Furthermore, we have experimented with different tile sizes ($w \times h \times t$), and have arrived at two resonable sizes, namely, $4 \times 4 \times 4$ and $8 \times 8 \times 4$ samples per tile. We use these tile sizes both for $m = 4$ and $m = 16$ samples per pixel (SPP) rates. For $m = 16$, this means that 4×4 pixels would need $4 \times 4 \times 16$ samples, i.e., four $4 \times 4 \times 4$ tiles. Hence, the samples in a 4×4 pixel region

are split along the time dimension. For $4 \times 4 \times 4$ tiles, the number of samples sums to $n_s = 64$ samples per tile, which fits in 4 cache lines in uncompressed form. Similarly, a tile of size of $8 \times 8 \times 4$ has $n_s = 256$, which occupies 16 cache lines in uncompressed form.

For each tile, we store 64 bits of tile table header information, which includes z_{min} and z_{max} and various mode bits. The involved memory bandwidth usage for this data is typically around a few percent of the total depth memory bandwidth usage, and so this layout has been left mostly unoptimized. In our results section, we will compare to the depth offset compression algorithm (see Section 2.2), and also to the combination of depth offset and our algorithm, since their strengths are somewhat complementary. The tile table header bit layouts for depth offset, our algorithm, and their combination are shown below.

	n_s	N	Mode bits 1	Mode bits 2	z_{min}/z_{max}
Depth offset	64	2	0	0	31 + 31
	256	4	0	0	30 + 30
Our algorithm	64	2	2	2	29 + 29
	256	4	2	2	28 + 28
Combination	64	2	2	2	29 + 29
	256	4	2	2	28 + 28

Here, N is a two or four-bit number that indicates how many cache lines that the current tile has been compressed to, where $N = 0$ indicates that the tile is uncompressed. Hence, if the value of N is 8 (and $n_s = 256$), the tile has been compressed down to 50% of its original size, for example. As can be seen in the table above, the depth offset algorithm uses its remaining tile table header bits to encode z_{min} and z_{max} at 31 or 30 bits each. For our algorithm, we store N as done for depth offset, and then we store two bits for the first layer, where 00 indicates that all the samples in the tile are cleared, 01 indicates that the layer is compressed using mode 0 ($Patch(x, y)$), 10 indicates mode 1 ($Plane(x, y, t)$), and 11 indicates mode 2 ($Patch(x, y, t)$). These modes are described in Section 3.2. The second two mode bits are used to describe layer 2, where 00 indicates that there is no second layer, and the rest is the same as for mode bits 1. This leaves 29 or 28 bits for each of z_{min} and z_{max}, which we have found to be sufficient precision for all our test scenes.

For the combination of the two algorithms, we can use the header layout from our algorithm with a small modification. If the *Mode bits 1* are 00, either we have a cleared tile *or* a tile compressed with depth offset, and this is determined by *Mode bits 2* being set to 00 for a cleared tile and 01 for depth offset. In our algorithm, the *Mode bits 2* are unused if *Mode bits 1* are 00, and therefore, we simply exploit unused bit combinations to fit depth offset compression into our tile header. This leaves us the same number of z_{min} and z_{max} bits as for our algorithm.

In our rasterizer, the z_{min} value for a tile is updated each time a sample depth, which is smaller than the current z_{min}, passes the depth test. The z_{max} value, on the other hand, requires that all samples are compared, and is thus updated only when the tile is evicted from the tile cache, prior to compression. We employ a tile cache of 64 kB and a tile table cache of 4 kB.

In the following, we describe the format of a compressed tile. The predictor function coefficients (a, b, c, etc) are stored in 32 bits each. For mode 0 and 1, this sums to 128 bits, while for mode 2, it amounts to 256 bits. When a single layer is used to compress a tile, we therefore need either $p = 128$ or $p = 256$ bits to encode the predictor. This leaves $k = \lfloor \frac{N \cdot 512 - p}{n_s} \rfloor$ bits for the residual, δ, for each sample.

If the tile contains two layers, two predictor functions must be en-

| Airship | Cannon | Spiders | Stone Giant | Spheres |

Figure 5: *Images from the test scenes used to generate our results. The images are taken from the Heaven benchmark and Stone Giant demo and feature highly tessellated geometry. The Spheres scene is a synthetic benchmark scene intended to stress test highly varying motion. The top row contains reference screenshots without motion (except for the Spheres scene which includes motion), and the bottom row shows depth buffers, rendered using our rasterization framework, including motion blur. All images were rendered at 1920×1080 pixels resolution. The images Airship and Cannon images are courtesy of Unigine Corp., and the Spiders and Stone Giant images are courtesy of BitSquid.*

coded. The cost, denoted by p_1 and p_2, of each of these is either 128 or 256 bits. In addition, an extra bit per sample for the layer index must be stored, which costs n_s bits. Thus, for the two-layer mode, we have $k = \lfloor \frac{N \cdot 512 - n_s - \sum p_i}{n_s} \rfloor$ bits for the per-sample residual, δ. We want to minimize the number of cache lines, N, needed to compress each tile in order to reduce memory bandwidth usage as much as possible. The value of N is calculated as follows:

$$N = \left\lceil \left(n_s \underbrace{\max_i k_i}_{\mathbf{M}} + \underbrace{n_s(n_l - 1)}_{\mathbf{I}} + \underbrace{\sum p_i}_{\mathbf{P}} \right) / 512 \right\rceil, \quad (6)$$

where $n_l \in \{1, 2\}$ is the number of layers. The $\mathbf{M}$ term is the maximum correction bits needed for any layer, the $\mathbf{P}$ term is the sum of the predictor coefficients, and $\mathbf{I}$ is the predictor index (only used for two layers). The division by 512 is to convert the number of bits into number of cache lines. In our implementation, we simply evaluate N for all compression combinations and pick the best one.

5 Results

Using the test scenes in Figure 5, we evaluated *depth offset* compression, which is described in Section 2.2, our algorithm, and the combination of our algorithm and depth offset. This combination is described in Section 4. We used tiles of either $4 \times 4 \times 4$ ($w \times h \times t$) or $8 \times 8 \times 4$ samples when measuring compression rates. This means that the time dimension is split into four groups for 16 samples per pixel. We do not claim to have found the optimal tile configurations, but they worked well for our scenes and memory system, and gave consistent results.

Compression ratios for the algorithms are presented in Figure 6, where compression ratio is defined as the total compressed z-bandwidth for a frame divided by the total uncompressed z-bandwidth. The best improvement occurs for the Cannon scene, with 4 samples per pixel and $8 \times 8 \times 4$ samples per tile. In this case, depth offset compresses down to about 75%, while our algorithm manages to compress down to 55%, which is a substantial difference. The combination of depth offset and our algorithm is consistently better than our algorithm, which implies that they are somewhat complementary. In general, the combination of the two algorithms improves upon depth offset compression by between 3–20%.

When compressing tiles with our algorithm, more than one mode could often be used to achieve the best compression. For mode 0,

this happens around 75% of the time, for mode 1 around 85%, and mode 2 lands on about 60%. 20–50% of the time (depending on scene and if one or two layers are used), a tile can be best compressed with one mode exclusively. Again, mode 1 is by far the most important, and can be used roughly 60–70% of the time to achieve the best compression, where as mode 0 is used for 10–30% of the tiles. Mode 2 is only used up to 5% of the time for $4 \times 4 \times 4$ tile sizes. For $8 \times 8 \times 4$ tiles, however, this figure raises slightly to 5–15%.

We present results for both $4 \times 4 \times 4$ and $8 \times 8 \times 4$ tile configurations. The reason for this is that, while our algorithms performs better on configurations with more samples in a tile, the larger tile sizes gave a larger overall bandwidth for the highly tessellated scenes (roughly 50% higher in our framework). The tile size of a hardware architecture will most likely be based on balancing a number of factors, such as the rasterizer, shader unit, memory controller, and so on. Therefore, the two configurations are presented in order to give a sense of how the compression efficiency scales with tile size.

In Figure 7, we study the performance of the algorithms as a function of increasing motion. As can be seen in the diagrams, our algorithm is substantially better than depth offset compression for small amounts of motion, and then for larger amounts of motion, the gap is somewhat reduced.

6 Discussion

We have presented an algorithm that preserves the benefits of traditional depth buffer plane encoding algorithms while also being robust for stochastically sampled depth data, which previous algorithms have not even attempted to target.

To generate our results, we opted to split the *time* dimension into different tiles when working with 16 samples per pixel. When motion grows large relative to the triangle size, the rasterizer access pattern will become increasingly random which, in turn, makes the frame buffer cache less efficient. It is therefore important to design the whole frame buffer and rasterizer with this in mind, and to find a good balance which gives low bandwidth for static scenes, while scaling gracefully with increasing motion. We leave a detailed study of this for future work.

In our research, we have taken a first step towards efficient depth buffer compression of motion blur renderings. However, we believe that our work can be improved further, and in a near future, we will investigate various optimizations of our basic algorithms. For future work, we also want to look into how our algorithms can be
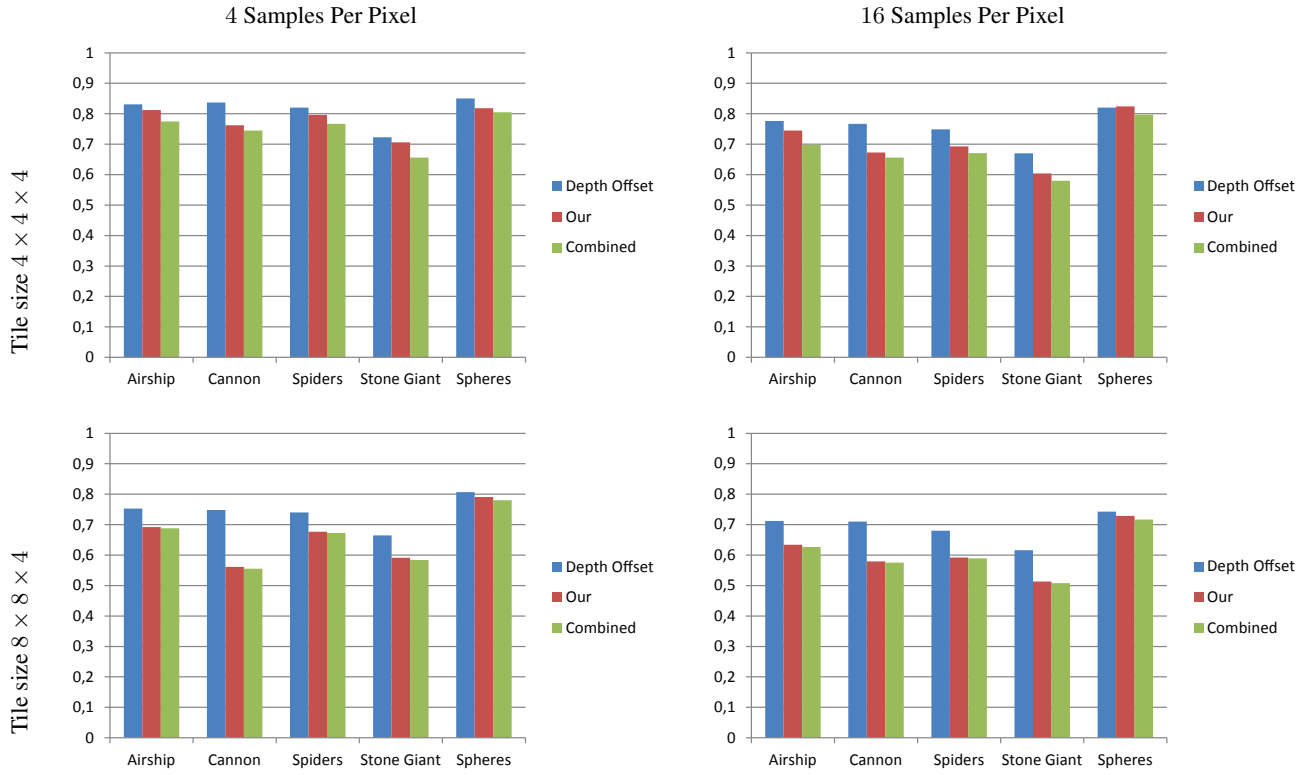
Figure 6: *Effective compression ratios for depth offset compression, our algorithm, and the combination of depth compression and our algorithm. We present compression results for 4 and 16 samples per pixel, as well as for two different tile configurations, namely,* $4 \times 4 \times 4$ *and* $8 \times 8 \times 4$.

extended to handling depth of field, as well as the combination of motion blur and depth of field at the same time.

References

AKENINE-MÖLLER, T., MUNKBERG, J., AND HASSELGREN, J. 2007. Stochastic Rasterization using Time-Continuous Triangles. In *Graphics Hardware*, 7–16.

BRUNHAVER, J., FATAHALIAN, K., AND HANRAHAN, P. 2010. Hardware Implementation of Micropolygon Rasterization with Motion and Defocus Blur. In *High Performance Graphics*, 1–9.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The Reyes Image Rendering Architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, vol. 21, 95–102.

DALLY, W. 2009. Power Efficient Supercomputing. Accelerator-based Computing and Manycore Workshop (presentation).

FATAHALIAN, K., LUONG, E., BOULOS, S., AKELEY, K., MARK, W. R., AND HANRAHAN, P. 2009. Data-Parallel Rasterization of Micropolygons with Defocus and Motion Blur. In *High Performance Graphics*, 59–68.

GOLUB, G., AND LOAN, C. V. 1996. *Matrix Computations*, 3rd ed. John Hopkins University Press.

GRIBEL, C. J., DOGGETT, M., AND AKENINE-MÖLLER, T. 2010. Analytical Motion Blur Rasterization with Compression. In *High Performance Graphics*, 163–172.

HASSELGREN, J., AND AKENINE-MÖLLER, T. 2006. Efficient Depth Buffer Compression. In *Graphics Hardware*, 103–110.

HOULE, M., AND TOUSSAINT, G. 1988. Computing the Width of a Set. *IEEE Transactions on Pattern Analysis and Machine Intelligence, 10*, 5, 761 –765.

LLOYD, D. B., GOVINDARAJU, N. K., MOLNAR, S. E., AND MANOCHA, D. 2007. Practical Logarithmic Rasterization for Low-Error Shadow Maps. In *Graphics Hardware*, 17–24.

MCGUIRE, M., ENDERTON, E., SHIRLEY, P., AND LUEBKE, D. 2010. Real-Time Stochastic Rasterization on Conventional GPU Architectures. In *High Performance Graphics*, 173–182.

MOREIN, S. 2000. ATI Radeon HyperZ Technology. In *Workshop on Graphics Hardware, Hot3D Proceedings*, ACM Press.

SALVI, M., VIDIMCE, K., LAURITZEN, A., AND LEFOHN, A. 2010. Adaptive Volumetric Shadow Maps. *Computer Graphics Forum (Proceedings of EGSR), 29*, 4 (June), 1289–1296.

STRÖM, J., WENNERSTEN, P., RASMUSSON, J., HASSELGREN, J., MUNKBERG, J., CLARBERG, P., AND AKENINE-MÖLLER, T. 2008. Floating-Point Buffer Compression in a Unified Codec Architecture. In *Graphics Hardware*, 96–101.

TOTH, R., AND LINDER, E. 2008. *Stochastic Depth of Field using Hardware Accelerated Rasterization*. Master's thesis, Lund University.
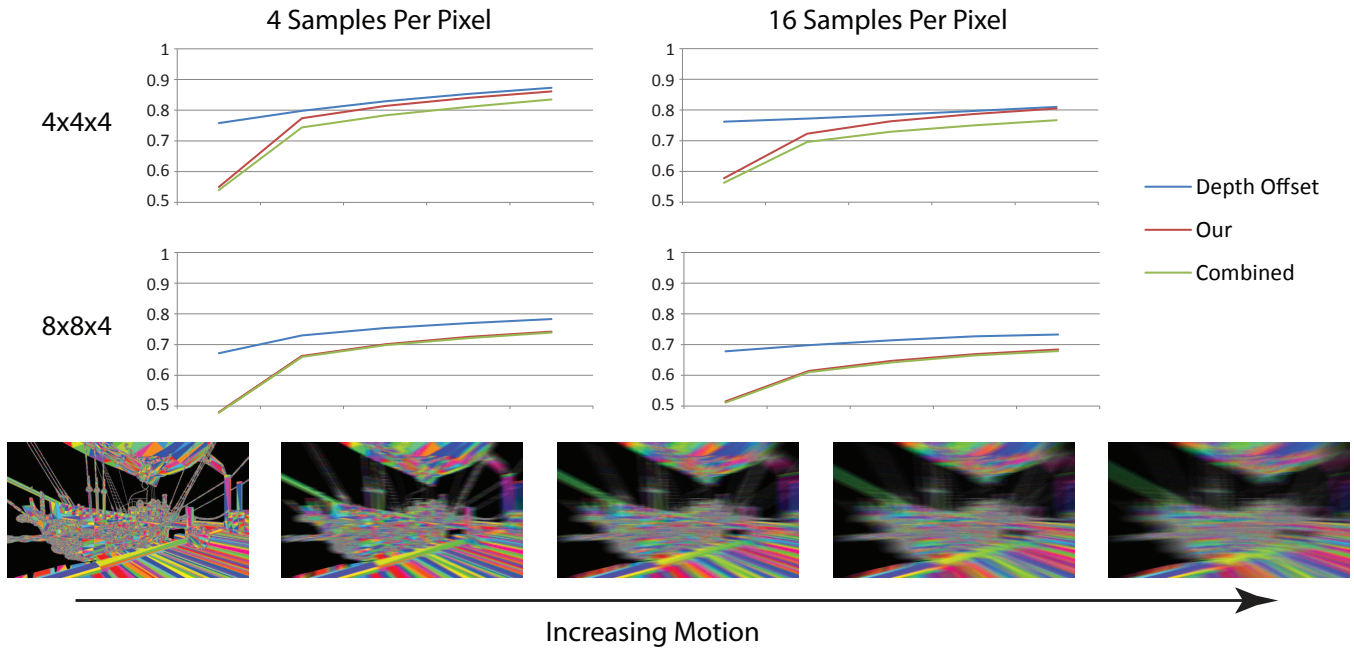
Figure 7: *Effective compression ratios for depth offset compression, our algorithm, and the combination of depth offset with our algorithm, with varying amounts of motion blur. Note that our algorithm performs better than depth offset across all amounts of motion for each configuration, and that the combination of the two algorithms is even a bit better for the $4 \times 4 \times 4$ tile configuration. In general, all three algorithms have stable performance with varying amounts of motion, which is in contrast to traditional planar encoders that typically break down when motion is introduced.*