Efficient Depth Buffer Compression

Jon Hasselgren Tomas Akenine-Möller

Lund University

Abstract

Depth buffer performance is crucial to modern graphics hardware. This has led to a large number of algorithms for reducing the depth buffer bandwidth. Unfortunately, these have mostly remained documented only in the form of patents. Therefore, we present a survey on the design space of efficient depth buffer implementations. In addition, we describe our novel depth buffer compression algorithm, which gives very high compression ratios.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Picture/Image Generation]: framebuffer operations

1. Introduction

The depth buffer was originally invented by Ed Catmull, but first mentioned by Sutherland et al. [SSS74] in 1974. At that time it was considered a naive brute force solution, but now it is the de-facto standard in essentially all commercial graphics hardware, primarily due to rapid increase in memory capacity and low memory cost.

A naive implementation requires huge amounts of memory bandwidth. Furthermore, it is not efficient to read depth values one by one, since a wide memory bus or burst accesses can greatly increase the available memory bandwidth. Because of this, several improvements to the depth buffer algorithm have been made. These include: the tiled depth buffer, depth caching, tile tables [MWY03], fast z-clears [Mor00], z-min culling [AMS03], z-max culling [GKM93, Mor00], and depth buffer compression [MWY03]. A schematic illustration of a modern architecture implementing all these features is shown in Figure 1.

Many of the depth buffer algorithms mentioned above have never been thoroughly described, and only exist in textual form as patents. In this paper, we attempt to remedy this by presenting a survey of the modern depth buffer architecture, and the current depth compression algorithms. This is done in Section 2 & 3, which can be considered previous work. In Section 4 & 5, we present our novel depth compression algorithm, and thoroughly evaluate it by comparing it to our own implementations of the algorithms from Section 3.

2. Architecture Overview

A schematic overview implementing several different algorithms for reducing depth buffer bandwidth usage is shown in Figure 1. Next, we describe how the depth buffer collaborates with the other parts of a graphics hardware architecture.

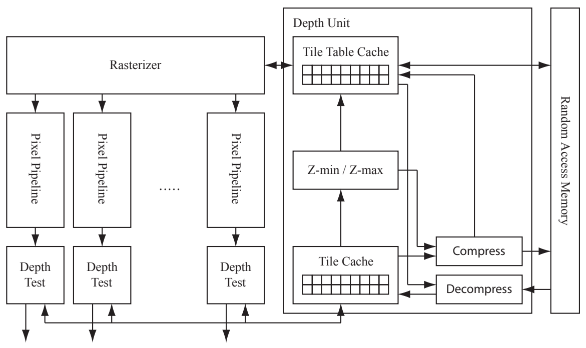


Figure 1: *A modern depth buffer architecture. Only the tile cache is needed to implement tiled depth buffering. The rest of the architecture is dedicated to bandwidth and performance optimizations. For a detailed description see Section 2.*

The purpose of the *rasterizer* is to identify which pixels lie within the triangle currently being rendered. In order to maximize memory coherency for the rest of the architecture, it is often beneficial to first identify which *tiles* (a collection of $n \times m$ pixels) that overlap the triangle. When the rasterizer finds a tile that partially overlaps the triangle, it distributes the pixels in that tile over a number of *pixel pipelines*. The purpose of each pixel pipeline is to compute the depth and color of a pixel. Each pixel pipeline contains a *depth test* unit responsible for discarding pixels that are occluded by previously drawn geometry.

Tiled depth buffering in its most simple form works by letting the rasterizer read a complete tile of depth values from the depth buffer and temporarily store it in on-chip memory. The depth test in the pixel pipelines can then simply compare the depth value of the currently generated pixel with the value in the locally stored tile. In order to increase overall performance, it is often motivated to cache more than one tile of depth buffer values in on-chip memory. A costly memory access can be skipped altogether if a tile already exists in the cache. The tiled architecture decrease the number of memory accesses, while increasing the size of each access. This is desirable since bursting makes it more efficient to write big chunks of localized data.

There are several techniques to improve the performance of a tiled depth buffer. A common factor for most of them is that they require some form of "header" information for each tile. Therefore, it is customary to use a *tile table* where the header information is kept separately from the depth buffer data. Ideally, the entire tile table is kept in on-chip memory, but it is more likely that it is stored in external memory and accessed through a cache. The cache is then typically organized in *super-tiles* (a tile consisting of tiles) in order to increase the size of each memory access to the tile table. Each tile table entry typically contains a number of "flag" bits, and potentially the minimum and maximum depth values of the corresponding tile.

The maximum and minimum depth values stored in the tile table can be used as a base for different culling algorithms. Culling mainly comes in two forms: z-max [GKM93, Mor00] and z-min [AMS03]. Z-max culling uses a conservative test to detect when all pixels in a tile are guaranteed to fail the depth test. In such a case, we can discard the tile already in the rasterizer stage of the pipeline, yielding higher performance. We can also avoid reading the depth buffer, since we already know that all depth tests will fail. Similarly, Z-min culling performs a conservative test to determine if all pixels in a tile are guaranteed to pass the depth tests. If this holds true, and the tile is entirely covered by the triangle currently being rendered, then we know that all depth values will be overwritten. Therefore we can simply clear an entry in the depth cache, and need not read the depth buffer.

The flag bits in the tile table are used primarily to flag different modes of depth buffer compression. A modern depth buffer architecture usually implements one or several compression algorithms, or compressors. A compressor will, in general, try to compress the tile to a fixed *bit rate*, and fails if it cannot represent the tile in the given number of bits without information loss. When writing a depth tile to memory, we select the compressor with the lowest bit rate, that succeeds in compressing the tile. The flags in the tile table are updated with an identifier unique to that compressor, and the compressed data is written to memory. We must write the tile in its uncompressed form if all available compressors fail, and it is therefore still necessary to allocate enough external memory to hold an uncompressed depth buffer. When a tile is read from memory, we simply read the compressor identifier from the tile table, and decompress the data using the corresponding decompression algorithm.

The main reason that depth compression algorithms can fail is that the depth compression must be lossless. The compression occurs each time a depth tile is written to memory, which happens on a highly unpredictable basis. Lossy compression amplifies the error each time a tile is compressed, and this could easily make the resulting image unrecognizable. Hence, lossy compression must be avoided.

3. Depth Buffer Compression - State of the Art

In this section, we describe existing compression algorithms. It should be emphasized that we have extracted the information below from patents, and that there may be variations of the algorithms that perform better, but such knowledge usually stays in the companies. However, we still believe that the general discussion of the algorithms is valuable.

A reasonable assumption is that each depth value is stored in 24 bits.[†] In general, the depth is assumed to hold a floating-point value in the range $[0.0, 1.0]$ after the projection matrix has applied. For hardware implementation, 0.0 is mapped to the 24-bit integer 0, and 1.0 is mapped to $2^{24}-1$. Hence, integer arithmetic can be used.

We define the term *compression probability* as the fraction of tiles that can be compressed by a given algorithm. It should be noted that the compression probability depends on the geometry being rendered, and can therefore only be determined experimentally.

3.1. Fast z-clears

Fast z-clears [Mor02] is a method that can be viewed as a simple form of compression algorithm. A flag combination in the tile table entry is reserved specifically for cleared tiles. When the hardware is instructed to clear the entire depth buffer, it will instead fill the tile table with entries that are flagged as cleared tiles. This means that the actual clearing process is greatly sped up, but it also has a positive effect when rendering geometry, since we need not read a depth tile that is flagged as cleared.

Fast z-clears is a popular compression algorithm since it gives good compression ratios and is very easy to implement.

3.2. Differential Differential Pulse Code Modulation

Differential differential pulse code modulation (DDPCM) [DMFW02] is a compression scheme, which exploits that the z-values are linearly interpolated in screen space. This algorithm is based on computing the second order depth differentials as shown in Figure 2. First, first-order differentials are computed columnwise. The procedure is repeated once again to compute the second-order columnwise differentials. Finally, the row-order

[†] Generalizing to other bit rates is straightforward.

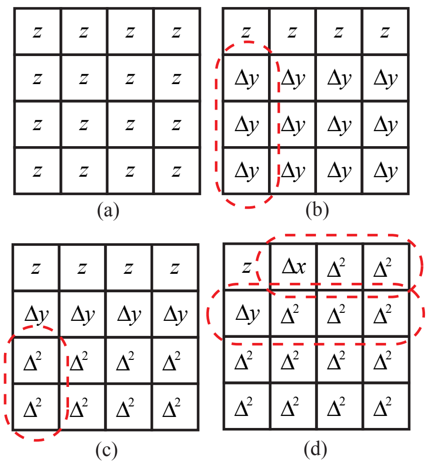


Figure 2: *Computing the second order differentials. a) Original tile, b) First order column differentials, c) Second order column differentials, d) Second order row differentials.*

differentials are computed for the two top rows, and we get the representation shown in Figure 2d. If a tile is completely covered by a single triangle, the second-order differentials will be zero, due to the linear interpolation. In practice, however, the second-order differential is a number in the set $\{-1,0,+1\}$ if depth values are interpolated at a higher precision than they are stored in, which often is the case.

DeRoo et al. [DMFW02] propose a compression scheme for 8×8 pixel tiles that use 32 bits for storing a reference value, 2×33 bits for x and y differentials, and 61×2 bits for storing the second order differential of each remaining pixel in the tile. This gives a total of 220 bits per tile in the best case (when a tile is entirely covered by a single triangle). A reasonable assumption would be that we read 256 bits from the memory, which would give a 8 : 1 compression when using a 32-bit depth buffer. Most of the other compression algorithms are designed for a 24-bit depth format, so we extend this format to 24 bit depth for the sake of consistency. In this case, we could sacrifice some precision by storing the differentials as 2×23 bits, and get a total of 192 bits per tile, which gives the same compression ratio as for the 32 bit mode.

In the scheme described above, two bits per pixel are used to represent the second order differential. However, we only need to represent the values: $\{-1,0,+1\}$. This leaves one bit-combination that can be used to flag when the second-order differential is outside the representable range. In that case, we can store a fixed number of second-order differentials in a higher resolution, and pick the next in order each time an escape code occurs. This can increase the compression probability somewhat at the cost of a higher bit rate.

DeRoo et al. also briefly describe an extension of the DDPCM algorithm that is capable of handling some cases of tiles containing two different planes separated by a single edge. They compute the second order differentials from two different reference points, the upper left and lower left pixels of the tile. From these two representations, one *break point* is determined along every column, such that pixels before and after the break point belong to different planes. The break points are then used to combine the two representations to a single representation. A 24-bit version of this mode would require $24\times6+2\times57+8\times4=290$ bits of storage.

The biggest drawback of the suggested two plane mode is that compression only works when the two reference points lie in different planes. This will only be true in half of the cases, if we assume that all orientation and positioning of the edge separating the two plane is equally probable.

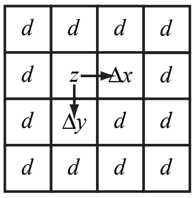


Figure 3: *Anchor encoding of a 4×4 tile. The depth values of the z, Δx and Δy pixels form a plane. Compression is achieved by using the plane as a predictor, and storing an offset, d, for each pixel. Only 5 bits are used to store the offsets.*

3.3. Anchor encoding

Van Dyke and Margeson [VM05] suggest a compression technique quite similar to the DDPCM scheme. The approach is based on 4×4 pixel tiles (although it could be generalized) and is illustrated in Figure 3. First, a fixed anchor pixel, denoted z in the figure, is selected. The depth value of the anchor pixel is always stored at full 24-bit resolution. Two more depth values, Δx and Δy, are stored relatively to the depth value of the anchor pixel, each with 15 bits of resolution. These three values form a plane, which can be used to predict the depth values of the remaining pixels. Compression is achieved by storing the difference between the predicted, and actual depth value, for the remaining pixel. The scheme uses 5 bits of resolution for each pixel, resulting in a total of 119 bits (128 with a fast clear flag and a constant stencil value for the whole tile).

The anchor encoding mode behaves quite similar to the one plane mode of the DDPCM algorithm. The extra bits of per-pixel resolution provide for some extra numerical stability, but unfortunately do not seem to provide a significant increase in terms of compression ratio.

3.4. Plane Encoding

The previously described algorithms use a plane to predict the depth value of a pixel, and then correct the prediction using additional information. Another approach is to skip

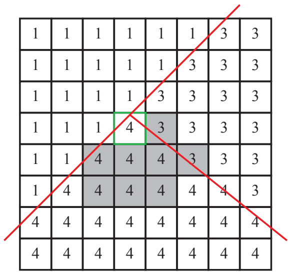


Figure 4: *Van Hook's plane encoding uses ID numbers and the rasterizer to generate a mask indicating which pixels belong to a certain triangle. The compression is done by finding the first pixel with a particular ID and searching a window of nearby pixels, shown in gray, to compute a plane representation for all pixels with that ID.*

the correction factors and only store parameterized prediction planes. This only works when the prediction planes are stored in the same resolution that is used for the interpolation.

Orenstein et al. [OPS*05] present such a compression scheme, where a single plane is stored per 4×4 pixel tile. They use a representation on the form $Z(x,y) = C_0 + xC_x + yC_y$ with 40 bits of precision for each constant. A total of 120 bits is needed, leaving 8 bits for a stencil value. Exactly how the constants are computed, is not detailed. However, it is likely that they are obtained directly from the interpolation unit of the rasterizer. Computing high resolution plane constants from a set of low resolution depth values is not trivial.

A similar scheme is suggested by Van Hook [Van03], but they assume that the same precision (16, 24 or 32 bits) is used for storing and interpolating the depth values. The compression scheme can be seen as an extension of Orenstein's scheme, since it is able to handle several planes. It requires communication between the rasterizer and the compression algorithm. A counter is maintained for every tile cache entry. The counter is incremented whenever rasterization of a new triangle generates pixels in the tile, and each generated pixel will be tagged with that value as an identifier, as shown in Figure 4. The counter is usually given a limited resolution (4 bits is suggested) and if the counter overflows, no compression can be made. When a cache entry is compressed and written to memory, the first pixel with a particular ID number is found. This pixel is used as a reference point for the plane equation. The x and y differentials are found by searching the pixels in a small window around the reference point. Van Hook shows empirically that a window such as the one shown in Figure 4 is sufficient to be able to compute plane equations in 96% of the cases that could be handled with an infinite size window (tests are only performed on a simple torus scene though). The suggested compression modes stores a number of planes (2,4, or 8 with 24 bits per component) and an identifier for each pixel, indicating to which

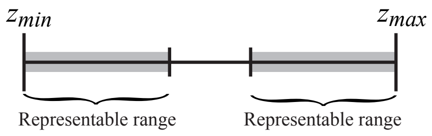


Figure 5: *The depth offset scheme compresses the depth data by storing depth values in the gray regions as offsets relative to either the z-min or z-max value.*

plane that pixel belongs (1,2 or 3 bits depending on the number of planes), resulting in compression ratios varying from $6:1$ to $2:1$. The compression procedure will automatically collapse any pixel ID numbers that is not currently in use. ID numbers may go to waste as depth values are overwritten when the depth test succeeds. Therefore, collapsing is important in order to avoid overflow of the ID counter. When decompressing a tile, the ID counter is initialized to the number of planes that is indicated by the compression mode.

The strength of the Van Hook scheme is that it can handle a large number of triangles overlapping a single tile, which is an important feature when working with large tiles. A drawback is that we must also store the 4-bit ID numbers, and the counter, in the depth tile cache. This will increase the cache size by $4/24 = 16.6\%$, if we use a 4-bit ID number per pixel. Another weakness is that the depth interpolation must be done at the same resolution as the depth values are stored in.

3.5. Depth Offset Compression

Morein and Natale's [MN04] depth offset compression scheme is illustrated in Figure 5. Although the patent is written in a more general fashion, the figure illustrates its primary use. The depth offset compression scheme assumes that the depth values in a tile often lie in a narrow interval near either the z-min value or the z-max value. We can compress such data by storing an n-bit offset value for every depth value, where n is some pre-determined number (typically 8 or 12) of bits. The most significant bit indicates whether the depth value is encoded as an offset relative to the z-min or z-max value, and the remaining bits represents the offset. The compression fails if the depth offset value of any pixel in a tile cannot be represented without loss in the given number of bits.

This algorithm is particularly useful if we already store the z-min and z-max values in the tile table for culling purposes. Otherwise we must store the z-min and z-max values in the compressed data, which increase the bit rate somewhat.

Orenstein et al. [OPS*05] also present a compression algorithm that is essentially a subset of Morein and Natale's algorithm. It is intended to complement the plane encoding algorithm described in Section 3.4, but can also be implemented independently. The depth value of a reference pixel is stored along with offsets for the remaining pixels in the tile. This mode can be favorable in some cases if the z-min and z-max values are not available.

The advantage of depth offset compression is that compression is very inexpensive. It does not work very well at high compression ratios, but gives excellent compression probabilities at low compression rates. This makes it an excellent complementary algorithm to use for tiles that cannot be handled with specialized plane compression algorithms (Sections 3.2-3.4).

4. New Compression Algorithms

In this section, we present two modes of a new compression scheme. As most other schemes, we try to achieve compression by representing each tile as number of planes and predict the depth values of the pixels using these planes.

In the majority of cases, depth values are interpolated at a higher resolution than is used for storage, and this is what we assume for our algorithm. We believe that this is an important feature, especially in the case of homogeneous rasterizers where exact screen space interpolation can be difficult. Allowing higher precision interpolation allows for some extra robustness.

In the following we will motivate that we only need the integer differentials, and a one bit per pixel *correction term*, in order to be able to reconstruct a rasterized plane. During the rasterization process, the depth value of a pixel is given through linear interpolation. Given an origin (x_0, y_0, z_0) and the screen space differentials $(\frac{\Delta z}{\Delta x}, \frac{\Delta z}{\Delta x})$, we can write the interpolation equations as:

$$z(x,y) \;=\; z_0 + (x - x_0)\frac{\Delta z}{\Delta x} + (y - y_0)\frac{\Delta z}{\Delta y}. \qquad (1)$$

The equation can be incrementally evaluated by stepping in the x-direction (similar for y) by computing:

$$z(x+1,y) \;=\; z(x,y) + \frac{\Delta z}{\Delta x}. \qquad (2)$$

We can rewrite the differential of Equation 2 as a quotient and remainder part, as shown below:

$$\frac{\Delta z}{\Delta x} = \left\lfloor \frac{\Delta z}{\Delta x} \right\rfloor + \frac{r}{\Delta x}. \qquad (3)$$

Equation 2 can then be stepped through incrementally by adding the quotient, $\lfloor \frac{\Delta z}{\Delta x} \rfloor$, in each step, and by keeping track of the accumulated remainder, $\frac{r}{\Delta x}$. When the accumulated remainder exceeds one, it is propagated to the result. What this amounts to in terms of compression is that we can store the propagation of the remainder in one bit per pixel, as long as we are able find the differentials $(\lfloor \frac{\Delta z}{\Delta x} \rfloor, \lfloor \frac{\Delta z}{\Delta y} \rfloor)$. This reasoning has much in common with Bresenham's line algorithm.

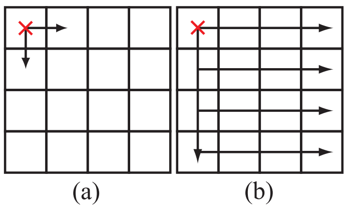
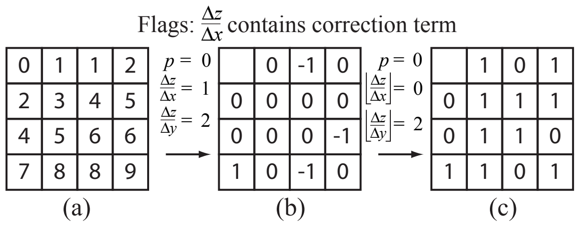


Figure 6: *The leftmost image shows the points used to compute our prediction plane. The rightmost image shows in what order we traverse the pixels of a tile.*


Flags: $\frac{\Delta z}{\Delta x}$ contains correction term
$p = 0$
$\frac{\Delta z}{\Delta x} = 1$
$\frac{\Delta z}{\Delta y} = 2$
$p = 0$
$\lfloor \frac{\Delta z}{\Delta x} \rfloor = 0$
$\lfloor \frac{\Delta z}{\Delta y} \rfloor = 2$
(a)
(b)
(c)

0	1	1	2
2	3	4	5
4	5	6	6
7	8	8	9

	0	-1	0
0	0	0	0
0	0	0	-1
1	0	-1	0

	1	0	1
0	1	1	1
0	1	1	0
1	1	0	1

Figure 7: *The different steps of the one plane compression algorithm, applied to a compressible example tile.*

4.1. One plane mode

For our one plane mode, we assume that the entire tile is covered by a single plane. We choose the upper left corner as a reference pixel and compute the differentials $(\frac{\Delta z}{\Delta x}, \frac{\Delta z}{\Delta y})$ directly from the neighbors in the x- and y-directions, as shown in Figure 6a. The result will be the integer terms,$(\lfloor \frac{\Delta z}{\Delta x} \rfloor, \lfloor \frac{\Delta z}{\Delta y} \rfloor)$, of the differentials, each with a potential correction term of one baked into it.

We then traverse the tile in the pattern shown in Figure 6b, and compute the correction terms based on either the x or y direction differentials (y direction when traversing the leftmost column, and x direction when traversing along a row). If the first non-zero correction term of a row or column is one, we flag that the corresponding differential as correct. Accordingly, if the first non-zero element is minus one, we flag that the differential contains a correction term. The flags are sticky, and can therefore only be set once. We also perform tests to make sure that each correction value is representable with one bit. If the test fails, the tile cannot be compressed.

After the previous step, we will have a representation like the one shown in Figure 7b. Just as in the figure, we can get correction terms of -1 for the differentials that contain an embedded correction term. Thus, we want to subtract one from the differential (e.g. $\frac{\Delta z}{\Delta x}$), and to compensate for this, we add one to all the per-pixel correction terms. Adding one to the correction terms is trivial since they can only be -1 or 0. We can just invert the last bit of the correction terms and interpret them as a one bit number. We get the corrected representation of Figure 7c.

In order to optimize our format, we wish to align the size

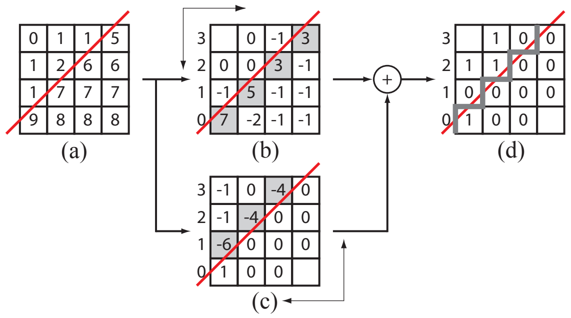


Figure 8: *This figure illustrates the two plane compression algorithm. a) Shows the original tile with depth values from two different planes. The line indicates the edge separating the two planes. b & c) We execute the one plane algorithm of Section 4.1 for each corner of the tile. In this figure, we only show the two correct corners for clarity. Note that the correction terms take on unrepresentable values when we cross the separating edge. We use this to detect the breakpoints, shown in gray. d) In a final step, we stitch together the two solutions from (b) and (c), and make sure to correct the differentials so that all correction terms are either 0 or 1. The breakpoints are marked as a gray line.*

of a compressed tile to the nearest power of two. In order to do so, we sacrifice some accuracy when storing the differentials, and reference point. Since the compression must be lossless, the effect is that the compression probability is slightly decreased, since the lower accuracy means that fewer tiles can be compressed successfully. Interestingly, storing the reference point at a lower resolution works quite well if we assume that the most significant bits are set to one. This is due to the non-linear distribution of the depth values. For instance, assume we use the projection model of OpenGL and have the near and far clip planes set to 1 and 100 respectively, then 21 bits will be enough to cover 93% of the representable depth range. In contrast, 21 bits can only represent 12.5% of the range representable by a 24 bit number. We propose the following formats for our one plane mode

tile	point	deltas	correction	total
4×4	21	14×2	1×15	64
8×8	24	20×2	1×63	127

4.2. Two plane mode

We also aim to compress tiles that contain two planes separated by a single edge. See Figure 8a for an example. In order to do so, we must first extend our one plane algorithm slightly. When we compute the correction terms, we already perform tests to determine if the correction term can be represented with one bit. If this is not the case, then we call the pixel a break point, as defined in Section 3.2, and store its horizontal coordinate. We only store the first such break point along each row. If a break point is found while traversing along a column, rather than a row, then all remaining rows are given a break point coordinate of zero. Figure 8b shows the break points and correction terms resulting from the tile in Figure 8a. As shown in the figure, we can use the break points to identify all pixels that belong to a specific plane.

We must also extend the one plane mode so that it can operate from any of the corners as reference point. This is a simple matter of reflecting the traversal scheme, from Figure 6, horizontally and/or vertically until the reference point is where we want it to be.

We can now use the extended one plane algorithm to compress tiles containing two planes. Since we have limited the algorithm to tiles with only a single separating edge, it is possible to find two diagonally placed corners of the tile that lie on opposite sides of the edge. There are only two configurations of diagonally placed corners, which makes the problem quite simple. The basic idea is to run the extended one plane algorithm for all four corners of the tile, and then find the configuration of diagonal corners for which the break points match. We then stitch together the correction terms of both corners, by using the break point coordinates. The result is shown in Figure 8d.

It should be noted that we need to impose a further restriction on the break points. Assume that we wish to recreate the depth value of a certain pixel, p, then we must be able to recreate the depth values of the pixels that lie "before" p in our fixed traversal order. In practice, this is not a problem since we are able to chose the other configuration of diagonal corners. However, we must perform an extra test. The break points must be either in falling or rising order, depending on which configuration of diagonal corners is used. As it turns out, we can actually use this to our advantage when designing the bit allocations for a tile. Since we know that the break points are in rising or falling order, we can use fewer bits for storing them. In our 4×4 tile mode, we use this to store the break points in just 7 bits. We do not use this in the 8×8 tile mode, as the logic would become too complicated. Instead, we store the break points using $\log_2(9^8) = 26$ bits, or with 4 bits per break point when possible.

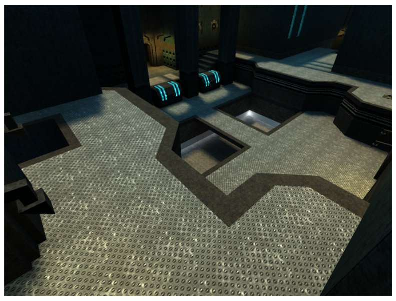
We employ the same kind of bit length optimizations as for the one plane mode. In addition, we need one bit, d, to indicate which diagonal configuration is used, and some bits for the break points, bp. Suggestions for bit allocations are shown in the following table.

tile	d	point	deltas	bp	correction	total
4×4	1	23×2	15×4	7	1×15	128
8×8	1	$22+21$	15×4	26	1×63	192
8×8	1	24×2	24×4	32	1×63	240

5. Evaluation

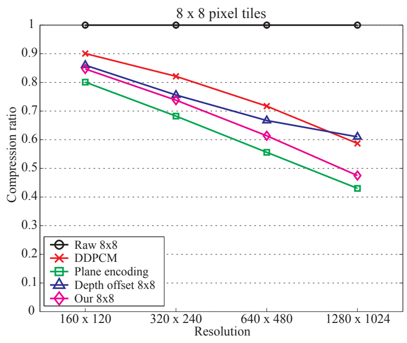
In this section, we compare the performance, in terms of bandwidth, of all depth compression algorithms described

Game Scene 1



Average #Pixels Per Triangle

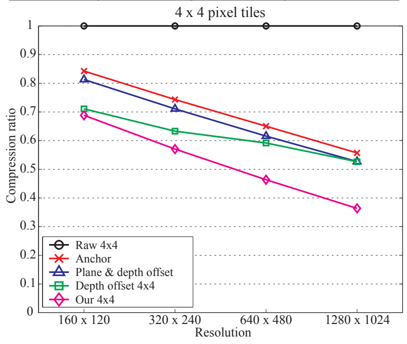
160 x 120	320 x 240	640 x 480	1280 x 1024
10.8	41.6	161.4	683.5



Game Scene 2

Average #Pixels Per Triangle

160 x 120	320 x 240	640 x 480	1280 x 1024
3.0	11.6	45.4	194.1



Sponza

Average #Pixels Per Triangle

160 x 120	320 x 240	640 x 480	1280 x 1024
0.6	2.4	9.0	37.6

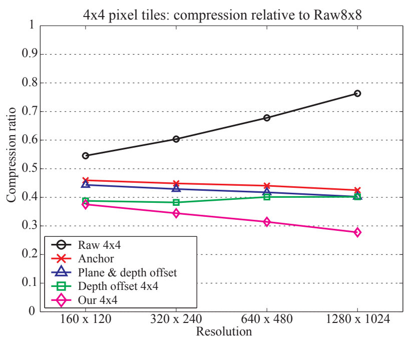


Figure 9: *The first row shows a summary of the benchmark scenes. The diagrams in the second row show the average compression for all three scenes as a function of rendering resolution, for* 4×4 *and* 8×8 *pixel tiles. Finally, we show the depth buffer bandwidth of* 4×4 *tiles, relative to the bandwidth of a Raw 8x8 depth buffer. It should be noted that this diagram does not take tile table bandwidth into account.*

in this paper. The tests were performed using our functional simulator, implementing a tiled rasterizer that traverses triangles a horizontal row of tiles at a time. We matched the tile size of the rasterizer to the tile size of each depth buffer implementation in order to maximize performance for all compression algorithms. Furthermore, we assumed a 64 bit wide memory bus, and accordingly, all our implementations of compressors have been optimized to make the size of all memory accesses aligned to 64 bits.

The depth buffer system in our functional simulator implements all features described in Section 2. We used a depth tile cache of approximately 2 kB, and full precision z-min and z-max culling. Our tests show that compression rates are only marginally affected by the cache size.[‡] Similarly, the z-min and z-max culling avoids a given fraction of the depth tile fetches, independent of compression algorithm. Therefore, it should affect all algorithms equally, and not affect the trend of the results.

Most of the compression algorithms have two operational modes. Therefore, we have chosen this as our target. Furthermore, two modes fit well into a two bit tile-table assuming we also need to flag for uncompressed tiles and for fast z clears. It is our opinion that using fast clears makes for a fair comparison of the algorithms. All algorithms can easily handle cleared tiles, which means that our compressors would be favored if this mode was excluded since they have the lowest bit rate.

We evaluate the following compression configurations

- **Raw 4x4/8x8**: No compression.
- **DDPCM**: The one and two-plane mode (not using “escape codes”) of the DDPCM compression scheme from Section 3.2, 8×8 pixel tiles. Bit rate: 3/5 bpp (bits per pixel)
- **Anchor**: The anchor encoding scheme (Section 3.3), 4×4 pixel tiles. Note that this is the only compression scheme in the test that only uses one compression mode. One bit-combination in the tile table was left unused. Bit rate: 8 bpp.
- **Plane encoding**: Van Hook’s plane encoding mode from section 3.4, 8×8 pixel tiles. Only the two and four plane modes were used, since we only allow 2 compression modes. This algorithm was given a slight favor in form of a 16.6% bigger depth tile cache. Bit rate: 4/7 bpp.

[‡] The efficiency of all algorithms increased slightly, and equally, with a bigger cache. We tested cache sizes of 0.5, 1, 2 and 4 kb

- **Plane & depth offset**: The plane (Section 3.4) and depth offset (Section 3.5) encoding modes of Orenstein et al, 4×4 pixel tiles. Bit rate: 8/16 bpp, 8 bits for the plane mode and 16 bits for the depth offset mode.
- **Depth Offset 4x4/8x8**: Morein and Natale's depth offset compression mode from Section 3.5. We used two compression modes, one using 12 bit offsets, and one with 16 bit offsets. Bit rate: 12/16 bits per pixel for both 4×4 and 8×8 tiles.
- **Our 4x4/8x8**: Our compression scheme, described in Section 4. For the 8×8 tile mode, we used the 192 bit version of the two plane mode in this evaluation. Bit rate: 4/8 bits per pixel for 4×4 tiles and 2/3 bits per pixel for 8×8 tiles.

Our benchmarks were performed on three different test scenes, depicted in Figure 9. Each test scene features an animated camera with static geometry. Furthermore, we rendered each scene at four different resolutions: $160 \times 120, 320 \times 240, 640 \times 480$, and 1280×1024 pixels. Varying the resolution is a simple way of simulating different levels of tessellation. As can be seen in Figure 9, we cover scenes with great diversity in the average triangle area.

In the bottom half of Figure 9, we show the compression ratio of each algorithm, grouped into algorithms for 4×4 and 8×8 pixel tiles. We also present the compression of the 4×4 tile algorithms, as compared to the bandwidth of the Raw 8x8 mode. It should be noted that this relative comparison only takes the depth buffer bandwidth into account. Thus, the bandwidth to the tile table will increase as the tile size decrease. How much of an effect this will have on the total bandwidth, will depend on the format of the tile table, and on the efficiency of the culling.

For 8×8 pixel tiles, our algorithm is the clear winner among the algorithms supporting high resolution interpolation, but it cannot quite compete with Van Hook's plane encoding algorithm. This is not very surprising considering that the plane encoding algorithm is favored by a slightly bigger depth tile cache, and avoids correction terms by imposing the restriction that depth values must be interpolated in the same resolution that is used for storage.

For 4×4 pixel tiles, the advantages of our algorithm becomes really clear. It is capable of bringing the two-plane flexibility that is only seen in the 8×8 tile algorithms down to 4×4 tiles, and still keeps a reasonably low bit rate. A two plane mode for 4×4 tiles is equal to having the flexibility of eight planes (with some restrictions) in an 8×8 pixel tile. This shows up in the evaluation, as our 4×4 tile compression modes have the best compression ratio at all resolutions.

6. Conclusions

We hope that our survey of previously existing depth buffer compression schemes will provide a valuable source for the graphics hardware community, as these algorithms have not been presented in an academic paper before. As we have shown, our new compression algorithm provides competitive compression for both 4×4 and 8×8 pixel tiles at various resolutions. We have avoided an exhaustive evaluation of whether 4×4 or 8×8 tiles provide better performance, since this is a very difficult undertaking which depends on several other parameters. Our work here has been mostly on an algorithmic level, and therefore, we leave more detailed hardware implementations for future work. We are certain that this is important, since such implementations may reveal other advantages and disadvantages of the algorithms. Furthermore, we would like to examine how to best deal with depth buffer compression of anti-aliased depth data.

Acknowledgements

We acknowledge support from the Swedish Foundation for Strategic Research and Vetenskapsrådet. Thanks for Jukka Arvo and Petri Nordlund of Bitboys for providing input.

References

[AMS03] AKENINE-MÖLLER T., STRÖM J.: Graphics for the Masses: A Hardware Rasterization Architecture for Mobile Phones. *ACM Transactions on Graphics, 22*, 3 (2003), 801–808.

[DMFW02] DEROO J., MOREIN S., FAVELA B., WRIGHT M.: Method and Apparatus for Compressing Parameter Values for Pixels in a Display Frame. In *US Patent 6,476,811* (2002).

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical Z-Buffer Visibility. In *Proceedings of ACM SIGGRAPH 93* (August 1993), ACM Press/ACM SIGGRAPH, New York, J. Kajiya, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, pp. 231–238.

[MN04] MOREIN S., NATALE M.: System, Method, and Apparatus for Compression of Video Data using Offset Values. In *US Patent 6,762,758* (2004).

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

[Mor02] MOREIN S.: Method and Apparatus for Efficient Clearing of Memory. In *US Patent 6,421,764* (2002).

[MWY03] MOREIN S., WRIGHT M., YEE K.: Method and apparatus for controlling compressed z information in a video graphics system. US Patent 6,636,226, 2003.

[OPS*05] ORNSTEIN D., PELED G., SPERBER Z., COHEN E., MALKA G.: Z-Compression Mechanism. In *US Patent 6,580,427* (2005).

[SSS74] SUTHERLAND E. E., SPROULL R. F., SCHUMACKER R. A.: A characterization of ten hidden-surface algorithms. *ACM Comput. Surv. 6*, 1 (1974), 1–55.

[Van03] VAN HOOK T.: Method and Apparatus for Compression and Decompression of Z Data. In *US Patent 6,630,933* (2003).

[VM05] VAN DYKE J., MARGESON J.: Method and Apparatus for Managing and Accessing Depth Data in a Computer Graphics System. In *US Patent 6,961,057* (2005).