Compression in the Graphics Pipeline

Mike Houston and Wei Koh

Computer Science Department
Stanford University

Abstract

This paper explores the subject of compression in the graphics pipeline. While there are several components of the pipeline that could stand to benefit from compression, we chose to focus on the Z-buffer and the framebuffer. We describe a method of compression using pixel differencing and Huffman coding along with a caching system designed around tiles. We examine the effects of tile size, pixel differencing schemes and variations of Huffman coding on compression rates, and show the tradeoffs between these parameters.

1 Introduction

As processors have gotten progressively faster, memory bandwidth has been unable to keep up. Some systems have turned to exotic forms of memory subsystems like bank-interleaving and custom designs like embedded DRAM to try to overcome bandwidth issues. Things are somewhat worse in the computer graphics industry because processors are currently getting faster at a higher rate than general purpose CPUs. It is becoming clear that one of the major problems in building faster graphics systems is that memory bandwidth is holding back the GPU. Because of the pipeline nature of graphics systems and the parallelism currently exploited in many systems, graphics hardware presents a unique system that has many different bandwidth issues and needs.

mhouston@graphics.stanford.edu kwkoh@cs.stanford.edu

Compression can be used in various stages of the graphics pipeline to help with this problem of limited bandwidth. We simulated a system that involved caching, tiling, and compression of the Z-buffer and framebuffer. By combining pixel differencing with Huffman coding, we achieved compression rates of up to 5:1.

1.1 Previous Work

Geometry: The newer NVIDIA boards have claimed triangle rates upwards of 30 million triangles per second. If the GPUs are fast enough, but the memory does not have the bandwidth to keep feeding the GPU, then why not use compression to try to increase the effective bandwidth? What if extensions or specialized hardware could be used to compress models sent to the graphics system? There has been some interesting research done on geometry compression, but none of the techniques are currently used in commodity hardware. The work done by Deering [1] and later continued by Chow [2] suggests interesting techniques for lossy geometry compression that yield good compression rates. Chow's compression schemes are now part of Java3D, using software compression/ decompression.

Z-Compression: Both the new ATI and NVIDIA boards both use some form of Z-compression to increase the speed of their graphics systems. They both claim about a 4:1 lossless compression ratio. ATI claims that Z-compression and Fast-Z clears can give them up to 20% performance increase in rendering rate.

Texture Compression: There has been lots of research in image compression in general, but few suggested methods have been applied to OpenGL systems. S3's S3TC and 3dfx's FXT1 are the only ones in regular use. They are lossy compression schemes and tend to create bad artifacts in certain situations. There is also an extension to 3D textures proposed by NVIDIA that is based on the S3TC scheme and has the same benefits and problems. The performance increase in Quake3 with the use of S3TC is of interesting note. The rendering rate (frames per second) sees upwards of a 25% performance jump with the use of compressed textures. This can be somewhat attributed to the fact that more of the textures can now reside on the graphics card and do not have to be loaded from main memory.

Framebuffer Compression: Not much research seems to have been done in this area.

2 Choice of Compression Scheme

Lossy/Lossless While lossy compression schemes have the potential to yield very high compression rates, they also have the potential to create unwanted artifacts. This is especially true with the Z-buffer, as even a small deviation from the actual value can result in the wrong object being displayed. Thus, we have decided to limit our choice of compression scheme to only lossless schemes.

Encode/Decode Speed When dealing with compression of the Z-buffer and framebuffer, it is important that we look at both the encoding as well as the decoding speed of the algorithm. Some algorithms, such as vector quantization, are relatively fast at decoding, but rather slow at encoding. We wanted an algorithm that performs well in both directions.

Tables/Codebooks Some compression algorithms require that a table or codebook be transmitted with the data for it to be decoded on the other end. Since we are operating on a tiling system, with tiles of size 4x4, 8x8, or 16x16, it is very possible that the size of the table or codebook would negate any gains in the compression of the data. We decided to only consider compression schemes that did not involve the transmission of a table or codebook.

3 Huffman Coding

The basic idea behind Huffman coding is that symbols that occur more frequently should be represented by shorter codewords than symbols that occur less frequently. In an ideal case, you would know the probabilities of all the symbols in a given data set that you wish to compress. This is usually done by first scanning the entire data set and building a table of probabilities for each symbol encountered. This table is then used to construct a new table containing all the symbols of the data set and their corresponding codewords. A common way of representing this table of codewords is by building a binary tree where external nodes represent the symbols of the data set. The Huffman code for any symbol can be constructed by traversing the tree from the root node to the leaf node. Each time a traversal is made to the left child, a '0' is added to the code, while a '1' is added for a traversal to the right. This Huffman tree is then passed to the decoder so that it can uncompress the encoded data.

As previously mentioned, due to the small size of our tiles, we want to avoid algorithms that have to transmit tables or codebooks along with the compressed data. Also, having to scan the data in two passes: once for collecting the statistics, and the second for encoding the data, would incur a significant performance hit. To avoid these problems, we decided to pursue two alternate Huffman coding schemes: Huffman coding using a fixed codebook, and adaptive Huffman coding.

3.1 Fixed Codebook Huffman Coding

A simple way of not having to preprocess the data set is by using a fixed codebook that both the encoder and decoder units have access to at all times. This makes the procedure a one-pass process, and removes the need for the encoder to transmit the codebook to the decoder. The drawback is that the codebook will have to be general and will not be optimized for each data set, and so compression rates will not be as good as with an optimized codebook. It will also be a challenge to generate a general codebook that would yield decent results over a wide range of input data.

3.2 Adaptive Huffman Coding

In adaptive Huffman coding, the codebook is built and continually updated as the data is being encoded. The same happens in the decode stage. The decoder will build an identical codebook as it reads in the compressed data and uncompresses it. A binary code tree is used, similar to that described above. In addition to just storing the symbols at the leaves of the tree, an adaptive Huffman code tree also stores weights at each node, which correspond to the number of times that particular node has been traversed. Leaf nodes store the number of times each symbol has been encountered, while interior nodes store the sum of its two children. As more and more nodes are added, the weights are used to restructure the tree such that symbols that occur more frequently will have shorter codes than those that occur less frequently.

Adaptive Huffman coding utilizes a Not Yet Transmitted (NYT) node to specify that the symbol in the compressed stream has not yet been transmitted. Both the encoder and decoder will start with a tree that contains a single NYT node. When a new symbol is encountered, the encoder will transmit the NYT code, along with the uncompressed, or fixed code for that symbol. It then adds the new symbol to the table, and the next time it encounters that symbol, it can simply use the current code for that symbol and update the weights for the affected nodes. The decoding process is very similar. If it reads the code for the NYT node, it knows that the following code will be in its uncompressed, or fixed state. It then adds that symbol to the tree, and when it sees that code for that symbol in the future, it knows how to decode it.

3.3 DPCM and DDPCM

Since image pixels are often heavily correlated with neighboring pixels, an effective technique to improve the compressibility of the input data is to store pixel differences across the data, instead of storing each pixel value individually. For example, a smoothly changing set of pixels such as {0 1 2… 255} would not compress will with just a simple Huffman coding scheme. However, if we instead store pixel differences across neighboring pixels, the data becomes a series of '1's, which will be highly compressible with Huffman coding. This differencing scheme forms the basis for differential pulse code modulation (DPCM). In its basic form, pixels are differenced across the rows of the image, and when the end of the row is reached, it is continued at the next row in one single pass. A variation on this technique would be to first take differences across the rows, then to take the differences along the columns, starting at the top and moving down an image. This technique is known as differential DPCM, or DDPCM.

4 Tiling and Caching

Rather than compressing and decompressing the entire framebuffer each time a pixel is written or read, our proposed system will divide the framebuffer into equally sized square tiles. To further minimize the need to read from and write to the framebuffer, our system will incorporate a small cache for storing these tiles of uncompressed data. When a pixel is read from the framebuffer, our system will check if the tile that contains that pixel is currently stored in the cache. If so, it simply reads the value from the cache. If not, it loads the compressed version of the tile containing that pixel from the framebuffer, decompresses it, and stores it in the cache. We decided to use a random replacement policy in the cache due to its simplicity, and also due to the fact that more exotic replacement schemes would be more costly to compute, and only increase our hit rate by an insignificant amount.

5 Testing Setup

To test the effectiveness of the proposed compression scheme, we used GLSim, an instrumented implementation of OpenGL written by Ian Buck and Kekoa Proudfoot, and GLTrace, a tracing package written by Proudfoot.

We used MacDonald's adaptive Huffman encoding library [6], implemented DPCM and DDPCM, and incorporated them into GLSim. We added a cache simulator into GLSim to that simulates the caching and tiling system described above and outputs a log of total accesses, cache hits and cache misses.
We tested cache sizes of 1, 2, 4, 8, 16, and 32kB, and tile sizes of 4x4, 8x8, and 16x16. We also used adaptive as well as fixed codebook Huffman coding, and using those in conjunction with DPCM,

DDPCM, and with the original image data (without taking any differences). We used 4 different GL traces to test the compression system: Atlantis, Quake 4, QuickTime VR, and a custom trace that writes to various parts of the framebuffer in a random fashion. To test the fixed codebook Huffman coding, we generated tables that were created by "learning" on the Atlantis, Quake 4 and QuickTime VR traces. We also generated separate tables for use with DPCM, DDPCM, without either differencing schemes, and with the 3 tile sizes we were testing with.

6 Results

6.1 Compression Ratios

Table 1 compares the compression ratios for different combinations of fixed/adaptive Huffman coding schemes, coupled with no differencing/DPCM/DDPCM, and tile sizes of 4x4, 8x8 and 16x16 for the color buffer. Compression ratios were calculated by averaging together the ratios for Atlantis, Quake 4, QTVR and a random access fill.

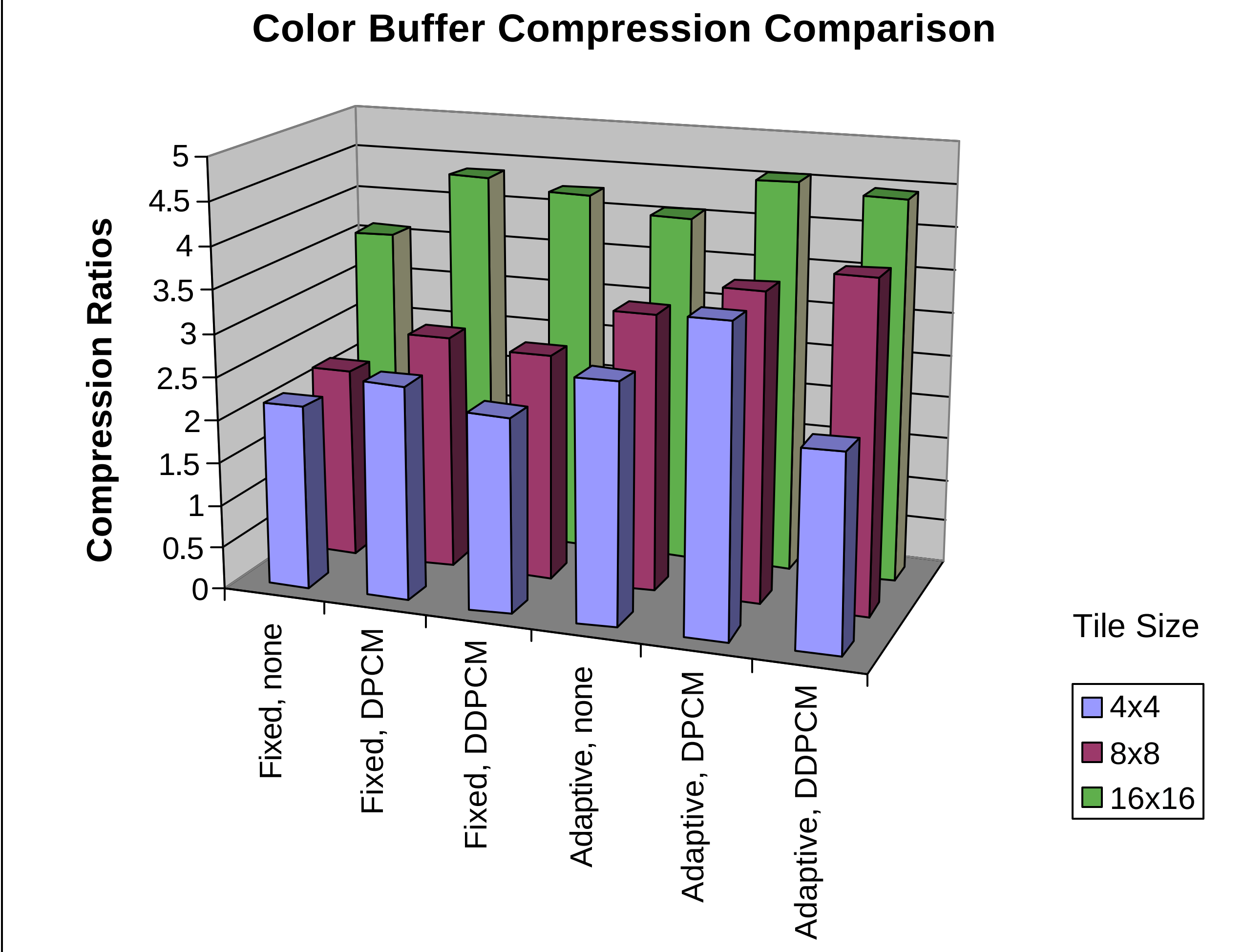

Table 1: Color Buffer Compression Ratios using an average of Atlantis, Quake 4, QTVR, and a random fill.

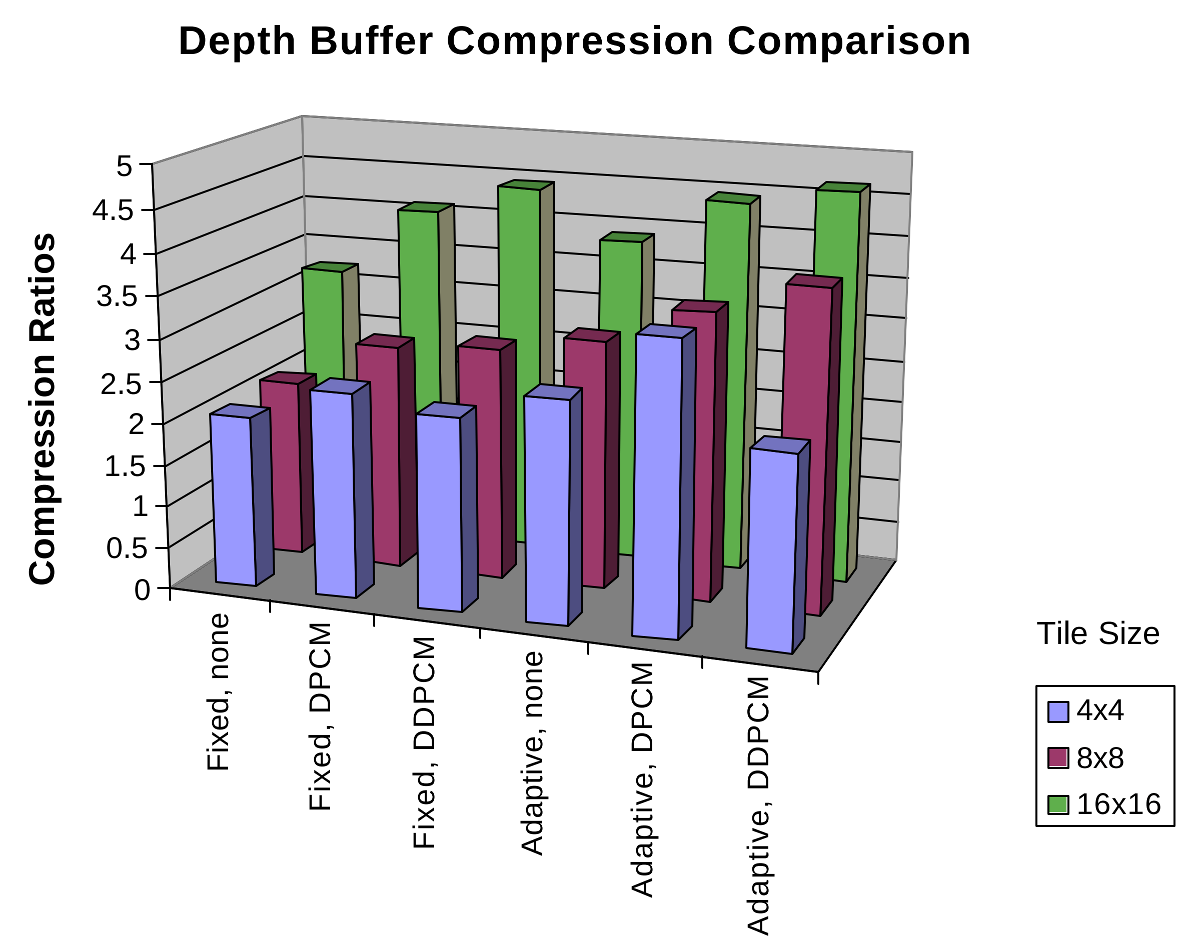

Table 2: Depth Buffer Compression Ratios using an average of Atlantis, Quake 4, QTVR, and a random fill.

		Depth Buffer									Color Buffer								
	Tile Size	**4x4**			**8x8**			**16x16**			**4x4**			**8x8**			**16x16**		
Huffman	**DPCM/DDPCM?**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**	**Avg**	**Min**	**Max**
Fixed	**None**	**2.22**	1.49	2.71	**2.32**	1.58	2.68	**3.63**	0.94	4.05	**2.24**	1.60	2.45	**2.36**	1.93	2.49	**3.54**	2.50	3.71
Adaptive	**None**	**2.46**	1.40	3.21	**3.33**	1.06	4.34	**4.37**	1.23	5.46	**2.45**	1.16	2.89	**3.27**	1.18	3.89	**4.12**	2.75	4.90
Fixed	**DPCM**	**2.52**	1.17	3.25	**2.80**	0.95	3.39	**4.75**	0.74	5.75	**2.69**	1.27	3.16	**2.97**	1.21	3.32	**5.20**	3.29	5.76
Adaptive	**DPCM**	**2.64**	1.28	3.57	**3.67**	1.22	4.82	**4.88**	1.42	5.91	**2.90**	1.18	3.47	**4.05**	1.31	4.80	**5.25**	3.05	5.91
Fixed	**DDPCM**	**2.28**	1.08	3.08	**2.72**	0.90	3.36	**4.66**	0.73	5.74	**2.37**	1.20	2.87	**2.84**	1.10	3.26	**5.04**	3.08	5.73
Adaptive	**DDPCM**	**2.34**	1.26	3.35	**3.48**	1.23	4.70	**4.78**	1.37	5.87	**2.51**	1.22	3.22	**3.78**	1.30	4.65	**5.10**	3.02	5.85

Table 3: Compression ratios for the Atlantis trace

As you can see, compression ratios increase with larger tile sizes, and with adaptive Huffman as compared to fixed codebook Huffman. DPCM definitely improves the compressibility of the data, but DDPCM further improves it only some of the time. The lowest compression ratio was 2.13:1, using a fixed codebook, no differencing, and a tile size of 4x4. The best compression ratio found was 4.57:1, using adaptive Huffman, DPCM and a tile size of 16x16.

Table 2 shows the results of our depth buffer testing. Compression ratios ranged from 2.05:1 (fixed codebook Huffman, no differencing, 4x4 tile size) to 4.65:1 (adaptive Huffman, DDPCM, 16x16 tile size). DPCM was always an improvement over no differencing, and DDPCM was a further improvement for tile sizes of 8x8 and 16x16.

A further breakdown of compression ratios can be seen in Table 3, which shows the results of running the Atlantis trace through various combinations of tile sizes (4x4, 8x8 and 16x16), fixed codebook/adaptive Huffman, and DPCM/DDPCM/ no differencing. Again, compression ratios are higher for adaptive Huffman when compared to fixed codebook Huffman, and increase as tile size increases. DPCM also increases compression ratios over not using any pixel differencing, but DDPCM does not seem to bring about any further increases in compression over DPCM. Results from testing with Quake 4 and QTVR were similar, with the exception that DDPCM did slightly better than DPCM about half the time, and slightly worse the other half

The results from our cache testing, as seen in Table 3, show that even a small amount of cache yields pretty decent hit rates. The trick is to balance the right amount of cache with the right tile size. While using a larger tile size increases the compressibility of the data, it also increases the amount of data that gets transferred with each cache miss. Table 3 shows the results of running the QTVR trace through various combinations of cache sizes (1, 2, 4, 8, 16 and 32KB) and tile sizes (4x4, 8x8 and 16x16). The break-even value is the compression ratio at which we break even on the bandwidth as compared to not using any compression.

As you can see, with a cache size of 8KB or less, going from a tile size of 8x8 to 16x16 requires a significantly higher compression ratio to break even. Since we get pretty decent hit rates with a cache size of 8KB and a reasonable break-even point for an 8x8 tile size, we can plug in our measured compression ratios for QTVR to calculate our savings in bandwidth. For fixed codebook Huffman, our best compression ratios for an 8x8 tile size are 2.83:1 for the depth buffer, using DDPCM, and 2.34:1 for the color buffer, using

Cache Size (KB)	Tile Size	Depth Cache			Color Cache		
		Hit	Miss	Break Even	Hit	Miss	Break Even
1	4	0.9002	0.0998	1.5975	0.9001	0.0999	1.5981
1	8	0.9522	0.0478	3.0569	0.9523	0.0477	3.0540
1	16	0.8894	0.1106	28.3256	0.8894	0.1106	28.3256
2	4	0.9093	0.0907	1.4506	0.9093	0.0907	1.4518
2	8	0.9640	0.0360	2.3039	0.9640	0.0360	2.3062
2	16	0.8900	0.1100	28.1544	0.8900	0.1100	28.1623
4	4	0.9188	0.0812	1.2999	0.9187	0.0813	1.3003
4	8	0.9705	0.0295	1.8867	0.9705	0.0295	1.8888
4	16	0.9837	0.0163	4.1702	0.9837	0.0163	4.1679
8	4	0.9266	0.0734	1.1745	0.9266	0.0734	1.1736
8	8	0.9769	0.0231	1.4796	0.9768	0.0232	1.4832
8	16	0.9890	0.0110	2.8278	0.9889	0.0111	2.8349
16	4	0.9314	0.0686	1.0969	0.9314	0.0686	1.0974
16	8	0.9805	0.0195	1.2471	0.9805	0.0195	1.2458
16	16	0.9929	0.0071	1.8071	0.9929	0.0071	1.8148
32	4	0.9340	0.0660	1.0553	0.9340	0.0660	1.0557
32	8	0.9823	0.0177	1.1334	0.9823	0.0177	1.1332
32	16	0.9945	0.0055	1.4124	0.9945	0.0055	1.4121

Table 4: Tradeoff between cache size and tile size for QTVR trace

DPCM. This results in a net reduction in bandwidth of 48% and 37% respectively. If adaptive Huffman is used, our maximum compression ratios are 3.44:1 for the depth buffer, using DDPCM, and 2.82:1 for the color buffer, using DPCM. These ratios yield a net reduction in bandwidth of 57% and 47% respectively.

7 Future Work and Considerations

7.1 Fixed codebook generation

Fixed codebooks would offer the simplest hardware implementation, but a learning system would have to be designed to gather data from many applications. It may be that designing a codebook to work with all applications would create a codebook that does not yield very good performance for any single application. Maybe a learning system could be designed to use several codebooks, and the hardware designed such that the best codebook is chosen based on the performance of the codebook used for the last rendered frame. DPCM and DDPCM seem to help a great deal with fixed codebooks since these systems are very good at evening out noise. As long as the data is not extremely noisy, these methods will generally generate lots of small values and many fewer large values. This means that the codebook can be designed to use shorter codewords for smaller values and larger codewords for larger values.

7.2 Other compression systems

There is a great deal of promise in the compression rates offered by other schemes. We did not include some of these systems based on the complexity of their implementations. However, if they can yield high enough compression rates to make up for their complexity, they might be able to get even higher bandwidth gains than the Huffman systems we chose to explore.

7.3 Detecting Low Compression Rates

One of the problems with any lossless compression system is that you can actually expand the data. In the case of our system, this can happen if the fixed table contains shorter codewords for low values, but the data contains mostly large values. A hardware design is needed that can detect that the compressed version is larger and can choose to send the original to the framebuffer. This would be a requirement for the hardware to work correctly, else depending on the alignment strategy used, it is likely that any data expansion will cause data in the framebuffer to be overwritten. This method would also potentially improve the average compression ratio if the method used occasionally causes data expansion.

7.4 Smarter Caches

The randomly drawn small polygons are a disastrous case for our system. The only way to prevent a large performance hit would be to not use compression if the miss rate takes us below the break even point for some conservatively calculated average compression ratio. It would be interesting to investigate the possible implementation of hardware the can detect such a situation and examine the consequences of such a design.

7.5 Texture Compression

Can a similar system to ours also be used in the texture stages of the pipeline? It would be interesting to see what the compression ratio and cache statistics would be if a system such as ours was applied to the texture memory. The framebuffer required both a fast encode and decode system, but textures are largely only read access. Although our method would most likely improve performance, it is more likely that methods that offer high compression rates with slow encodes and fast decodes would offer even better performance than our system. However, if something like our system was being used in hardware, it may only be a small addition to allow the same system to function with the texture units. Most likely, the best compression rates would be had with an algorithm, which provides high compression rates, inexpensive decode, and possibly expensive encoding.

8 Conclusion

As the performance increases in graphics processors continues to outpace the increases in the speed and bandwidth capabilities of memory systems, the use of compression will become more and more compelling. By using our system, it is possible for applications that are bound by the bandwidth to the framebuffer to potentially see a substantial increase in rendering performance. Even with very small caches, only containing a few tiles, it is still possible to yield a noticeable gain in performance. Since current graphics cards are currently using Z-compression and the numbers claimed by those manufacturers are similar to the numbers received by our methods and tests, we suggest that it would be possible to use similar hardware to that currently being used to also compress the color buffer. Based on our results, using a cache size of 8kB and a tile size of 8x8, it is possible to achieve close to a 50% improvement in bandwidth. If the bandwidth is holding the processor back, compression will give more room for the processor to run.

References

[1] DEERING, M. Geometry Compression. Computer Graphics (Proc. SIGRAPH), pages 101-108, August 1993.

[2] CHOW, M. M. Optimized Geometry Compression for Real-time Rendering, (Proc. IEEE VISUALIZATION '97), pages 347-354, 559.

[3] Texture Compression, from the Matrox website: http://www.matrox.com/mga/dev_relations/chip_tech/media/pdf/texture_compression.pdf

[4] BEERS, A.C., AGRAWALA, M., CHADDHA, N. Rendering from Compressed Textures, Computer Graphics (Proc. SIGGRAPH '96).

[5] SAYOOD, K. Introduction to Data Compression, Second Edition. Academic Press, 2000.

[6] Josh MacDonald. Adaptive Huffman Encoding Library. http://www.xcf.berkeley.edu/~ali/K0D/Algorithms/huff/