Compressed Multisampling for Efficient Hardware Edge Antialiasing

Philippe Beaudoin Pierre Poulin

LIGUM
Dép. I.R.O., Université de Montréal

Abstract

Today's hardware graphics accelerators incorporate techniques to antialias edges and minimize geometry-related sampling artifacts. Two such techniques, brute force supersampling and multisampling, increase the sampling rate by rasterizing the triangles in a larger antialiasing buffer that is then filtered down to the size of the framebuffer. The sampling rate is proportional to the number of subsamples in the antialiasing buffer and, when no compression is used, to the memory it occupies. In turn, a larger antialiasing buffer implies an increase in bandwidth, one of the limiting resources for today's applications. In this paper we propose a mechanism to compress the antialiasing buffer and limit the bandwidth requirements while maintaining higher sampling rates. The usual framebuffer-related functions of OpenGL are supported: alpha blending, stenciling, color operations, and color masking. The technique is scalable, allowing for user-specified maximal and minimal sampling rates. The compression scheme includes a mechanism to nicely degrade the quality when too much information would be required. A lower bound on the quality of the resulting image is also available since the sampling rate will never be less than the user-specified minimal rate. The compression scheme is simple enough to be incorporated into standard hardware graphics accelerators. Software simulations show that, for a given bandwidth, our technique offers improved visual results over multisampling schemes.

Key words: graphics hardware, edge antialiasing, multisampling

1 Introduction

Scan converting triangles is the core of today's hardware graphics accelerators. This process, which is really the act of discretizing a continuous signal, is usually performed by sampling triangles at the center of each pixel of the screen. We know that sampling can give rise to artifacts that are due to the presence of high frequencies in the continuous signal. In fact, the Nyquist formula tells us that such artifacts can occur as soon as the frequency of the input signal is greater than half the sampling rate.

Signals with such high frequencies are often witnessed in textures and that is why various methods have been designed to try to minimize texture-related sampling artifacts. The idea behind these techniques is to prefilter the input signal in order to remove all components above the Nyquist frequency. One such technique, trilinear mipmapping [17], is available on almost every hardware graphics accelerator.

Even though textures are usually the most important source of high frequencies in the input signal, the geometry itself can cause artifacts that will not be handled by texture filtering. That is because the triangle edges create discontinuities in the input signal resulting in arbitrarily high frequencies. The presence of such frequencies can create various artifacts. One of these is the staircase pattern visible along polygon edges. This problem, often referred to as "jaggies", is probably the most frequent geometry sampling artifact. However, Moiré patterns can also occur and tend to become more important as the size of triangles decreases.

Unfortunately, since the geometry-related input signal is not known in advance, prefiltering techniques such as mipmapping cannot be applied. In fact, the techniques available today in hardware do not filter out the components above the Nyquist frequency. Instead they increase the sampling rate, thus resulting in a higher Nyquist frequency leading to reduced artifacts.

With the usual techniques, a higher sampling rate results in an important increase in the internal bandwidth requirements of the graphics accelerator. Bandwith being one of the most limiting resources in today's applications, high sampling rates can result in an important performance degradation.

In this paper we present a technique to reduce the internal bandwidth requirements for a given sampling rate through the use of a compressed antialiasing buffer. This technique supports the usual framebuffer-related functions of OpenGL such as alpha blending, stenciling, color operations, and color masking. The compression scheme includes a mechanism to nicely degrade the quality when

too much information would be required. To do so, two sampling rates are provided by the user: a maximal rate used over most of the image and a minimal rate that limits its quality degradation. This minimal rate offers a lower bound on the quality of the resulting image, ensuring no erroneous results and reducing sampling artifacts.

The compression scheme is simple enough to be incorporated into standard hardware graphics accelerators at reasonable cost. Memory locations of neighboring pixels are close to one another, making the technique suitable to caching.

We have devised a software simulation of our technique and counted the number of memory accesses, which is a good measure of the required internal bandwidth. The software implementation is based on Mesa [20], an open source 3D library compatible with OpenGL. For a given bandwidth we have observed that our technique offers improved visual results over usual hardware methods.

This paper is organized as follows. First, we review previous work in software and hardware edge antialiasing. We then present our compressed antialiasing buffer and show how it helps reduce internal bandwidth requirements. After that, the software implementation and results are discussed. Finally, a conclusion and some future work are presented.

2 Previous Work

The two main techniques available today in hardware edge antialiasing are brute force supersampling and multisampling [13, 2]. A comprehensive survey can be found in [4]. The idea behind brute force supersampling is to render the geometry to an antialiasing buffer of larger resolution than the framebuffer. Once the entire scene has been rendered, the antialiasing buffer is filtered down to the size of the framebuffer. The increase in sampling rate is equal to the increase in resolution. A larger antialiasing buffer implies that more pixels will be generated by scan conversion, resulting in more texture look-ups, Z-reads, color-reads, Z-writes, and color-writes. All these operations require internal bandwidth which is one of the limiting resources in today's applications. In practice, supersampling often results in a performance drop.

Multisampling tries to limit the bandwidth requirements by sharing some color information among neighboring subsamples. This is made possible by the fact that textures are already filtered through trilinear mipmapping. Using multisampling, one can generate a number of fragments (polygons clipped by a pixel) approximately equal to the number of pixels in the framebuffer multiplied by the average depth complexity. Each fragment is associated to an array of subsamples stored in the antialiasing buffer. Therefore a fragment has a unique color, multiple depths, and a mask indicating which subsamples are covered. When updating the antialiasing buffer, only the subsamples indicated by the coverage mask are modified.

Since multisampling relies on texture prefiltering, the number of texture look-ups required is about the same as when rendering directly to the framebuffer: much less than required by supersampling. However, the number of Z-reads, color-reads, Z-writes, and color-writes are the same as supersampling. Therefore, multisampling still has bandwidth requirements that quickly increase with the sampling rate.

Other antialiasing approaches have also been implemented in hardware. The Accumulation Buffer [8] supersamples the scene by rendering it multiple times to a high precision framebuffer. Schilling [14] uses subpixel masks and edge orientation to evaluate pixel coverage and antialias edges. Deering and Naegle [7] perform supersampling using sparse non-repeating sampling patterns. Many authors [10, 15, 18, 19] have proposed variations on the A-buffer algorithm [6] better suited to hardware implementation. However, these techniques do not focus on minimizing bandwidth requirements.

More recently, Jouppi and Chang [9] propose to store each fragment with its depth and two Z gradients. Doing so, they devise a technique that gives good antialiasing results while storing only three fragments per pixel. Unfortunately, their approach can lead to undesirable results where completely occluded fragments contribute to the final image. A similar technique has been used by Matrox in its Parhelia graphics accelerator [11]. Lightweight multisampling schemes using 2 or even 1.25 sample per pixel have also been proposed [5, 3]. These techniques yield results similar in quality to the ones obtained with 4 samples per pixel but cannot be used to obtain higher quality images. Aila et al. [1] present a framebuffer-based algorithm to detect discontinuity edges. Antialiasing is then applied only to the pixels where discontinuity occurs. However, their algorithm relies on delay streams, a feature requiring important architecture modifications to current hardware designs.

3 Compressed Antialiasing Buffer

Our technique improves on earlier approaches through the use of a compressed antialiasing buffer. The main idea behind our compression scheme is to take advantage of the fact that multisampling generates a single color per fragment.

The antialiasing buffer will store color, depth, and stencil information. For the rest of this paper, we use 32-bit colors. We also consider that depth and stencil informa-

tions are combined within a 32-bit word, and refer to it simply as the depth.

3.1 Buffer Structure

Our compressed antialiasing buffer shares the same logical structure as the buffer used for standard multisampling. That is, all the subsamples are stored in a rectangular array. However, since multisampling generally produces a small number of colors per pixel, we store the colors as indices referring to entries in a color table.

In order to benefit from spatial coherency and to limit the size of indices, we partition the antialiasing buffer in sets of neighboring subsamples. Each set is contained within one pixel of the framebuffer and uses a single color table. Therefore, the indices used by subsamples in a set all refer to the same table. We shall refer to these sets of subsamples as the antialiasing buffer entries (Figure 1).

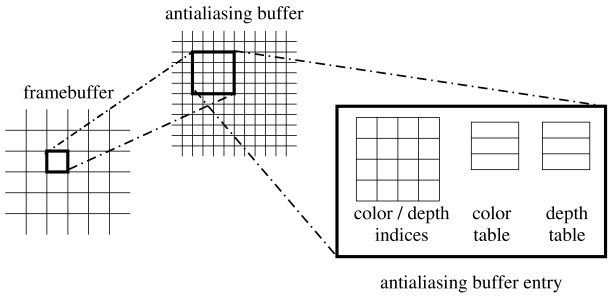

Figure 1: Structure of a single antialiasing buffer entry for a set of 4 subsamples.

We use a similar compression scheme for the depth information. However, since a fragment has more than one depth, using a table would not be efficient. To overcome this problem, we sample the depth of triangles once per fragment. This increases artifacts along the junction of interpenetrating surfaces, but allows our compression scheme to remain simple. In practice, edge aliasing due to interpenetrating polygons is usually negligible. The resulting depths are stored in the buffer entries using a table of the same size as the color table. Therefore, indices simultaneously refers to the color and depth tables.

For most fragments the depth is sampled at the center of the pixel. However, this can cause important artifacts when the fragment does not cover the pixel center. We avoid this problem by adjusting the sample location so that it remains within the fragment. We could adjust the color sample location in a similar fashion, but this was not required in practice.

Since we want to reduce the internal bandwidth required to transfer pixels, we will use tables that hold less colors and depths than the number of subsamples in an antialiasing buffer entry. The proposed technique can support tables of sizes ranging from one value per pixel to one value per subsample. For example, if a buffer entry contains 16 subsamples, we can use color and depth tables of size 1 to 16.

Holding an index to a table of sizes c requires $\lceil \log_2(c) \rceil$ bits. For the rest of the method to work correctly we require an extra index referring to an invalid subsample. Therefore, if we want to store c colors and depths per table, the number of bits B required for an antialiasing buffer entry containing s subsamples would be:

$$B(c, s) = s\lceil \log_2(c+1) \rceil + 64c. \quad (1)$$

3.2 Handling Overflow

It is possible to encounter a situation where more colors are required for a buffer entry than what we can fit in the table. We therefore need a way to handle color table overflow that lets us control image quality degradation.

The first solution would be to quantize colors when a table overflows. To do this, two colors are selected and replaced by their average color. Unfortunately, this can lead to severe artifacts due to the fact that the quantization has to be done during rendering without knowledge of future incoming fragments (Figure 2).

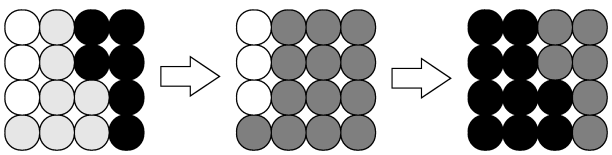

Figure 2: Possible error when color quantization is used to resolve overflow. In this example, the original pixel contains 3 colors while the color table can only hold 2. After quantization, the black and gray fragments are merged into a dark gray one. After the second black fragment is added, the pixel should be completely black, but it shows influence from the original gray fragment, now completely occluded.

Such problems can cause a completely occluded object to bleed through and contribute to the final image. In some applications, such as simulators or games, this effect can cause hidden information to become known to the user. For example, a player looking at a wall could know that something is moving behind it by looking for flickering pixels.

As mentioned by Jouppi and Chang [9], it is possible to reduce this artifact by selecting the colors of two subsamples having similar depths. However the problem cannot be completely eliminated since we can always encounter a buffer entry with greatly differing depths.

Moreover, we cannot directly average the depths of two subsamples, therefore this technique cannot be used

to handle depth table overflow. Jouppi and Chang [9] use the depth together with two extra Z gradients per fragment. They propose a technique to compute the new depth and Z gradients when merging two fragments. However, their technique will still produce artifacts when a pixel does not contain two fragments close in Z.

Instead of color quantization, we propose to locally reduce the sampling rate whenever the color or depth tables overflow. To do so, we hierarchically subdivide each antialiasing buffer entry (Figure 3). The last subdivision level represents the maximum sampling rate and contains all the subsamples of the antialiasing buffer entry. These subsamples are partitioned into groups that represent a lower sampling rate. This process is repeated until we reach one single group covering the entire buffer entry. In each of these groups we identify one subsample that will be the parent of its group. The result can be represented as a tree (Figure 4).

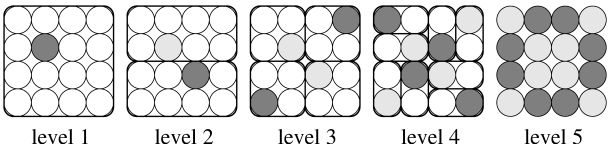

Figure 3: Example of hierarchical subdivision of an antialiasing buffer entry, parents of each group are shown in gray.

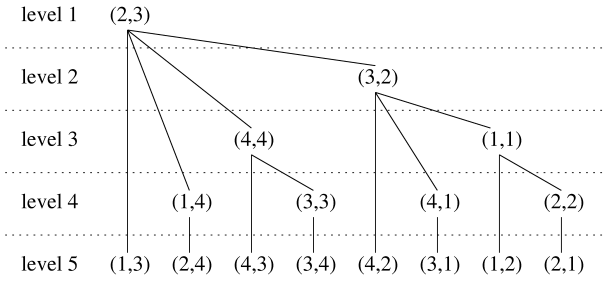

Figure 4: Tree containing all the subsamples of Figure 3. The subsamples are numbered starting from the lowest left corner.

3.3 Compression and Decompression

To perform depth tests, we need to evaluate the depth at each subsample of a pixel. This corresponds basically to decompressing a buffer entry. To do so, we could simply look up the depth table using this subsample's index. However, when an entry has been down-sampled, some subsamples may be using the invalid index mentioned in Section 3.1. To resolve invalid indices we need to perform a breadth-first traversal of the hierarchy.

We begin traversal at the second subdivision level and check if any of the nodes at this level are invalid. If we find such a node, we replace its index by that of its parent. Since we traverse the tree from top to bottom, and since our compression ensures that the top node is always valid, we know that this process will generate valid indices for each node.

This resolved array of indices will only be used to extract the depths of subsamples. All other operations are still performed on the original array. In particular, alpha blending is done using the original indices. We also define that blending any color with an invalid color results in an invalid color.

Inserting a new fragment into a pixel involves changing the color and depth of some subsamples. When doing so, we temporarily use as large a table and as many indices as we need so we do not have to worry about table overflow. We can therefore apply all the standard OpenGL fragment operations such as alpha blending, stenciling, color operations, and color masking. Complex pixel shaders can also be applied. After the final fragment has been computed, we need to generate a new compressed buffer entry and write it back to the antialiasing buffer.

Compression is performed using a breadth-first traversal of the hierarchy, counting the number of different indices encountered. Whenever we find a subsample having an index that would cause a table to overflow, we mark the index of this subsample as invalid. Moreover, whenever we generate or encounter an invalid node on level i, we mark as invalid its level $(i+1)$ child and the level $(i+1)$ child of its parent. For example, if we find node $(1,4)$ to be invalid in the tree of Figure 4, we mark nodes $(2,4)$ and $(1,3)$ as invalid.

This compression process makes sure that the top level node will never be invalid. Moreover, it ensures that whenever a subsample group cannot be completely valid, then *all* the subsamples in this group are invalid. The result of the execution of our compression and decompression process on the example of Figure 2 is shown in Figure 6. This technique correctly handles the incoming of the second black fragment.

At best our method will have results identical to standard multisampling. However, some scenes will cause a number of tables to overflow resulting in a loss of information. Fortunately it is possible to find a lower bound on quality degradation. To do so we notice that, given color and depth tables of size c, the first c nodes of the subsample tree will always be valid. Since these nodes are attached to distinct subsample groups that cover the whole pixel, we can assert that the worst case result is equal to standard multisampling with a lower resolution. Therefore, quality degradation depends on c and on the

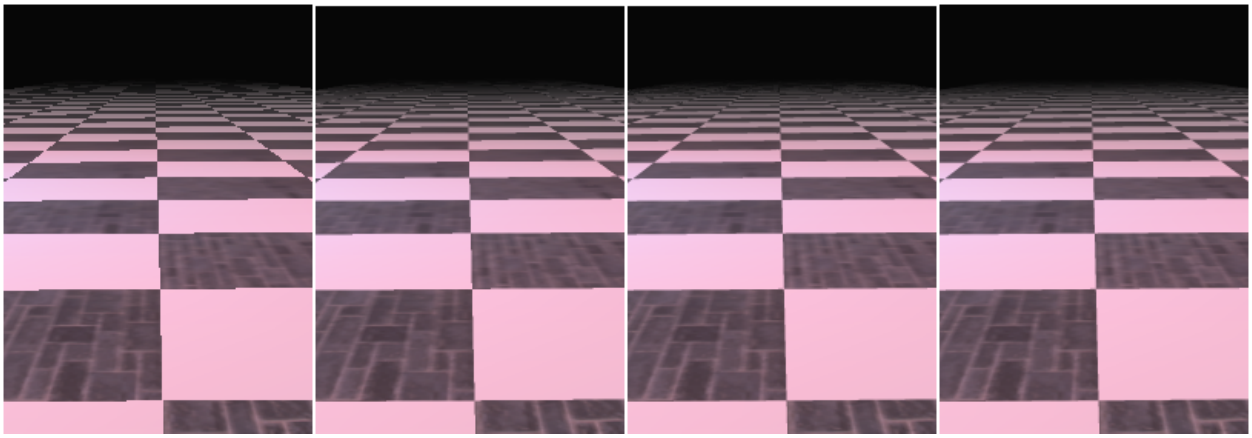

Figure 5: From left to right: no antialiasing, 2×2 multisampling, 4×4 compressed multisampling, and 4×4 multisampling.

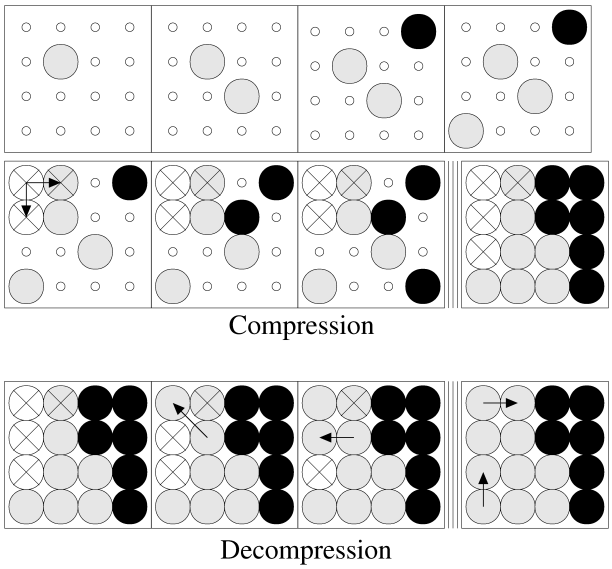

Figure 6: Compression and decompression of an overflowing color table for the pixel in Figure 2, invalid subsamples are crossed out. The antialiasing buffer entry can only hold 2 colors. We use the subsample hierarchy of Figure 4.

subsample hierarchy. For example, using the hierarchy of Figure 4 and tables of size 4 we know that the worst-case result will be identical to 4 sample per pixel sparse multisampling [16, 12].

4 Results

Our implementation is based on Mesa [20], an open source 3D library compatible with OpenGL. We modified the fragment processing code in order to support variable rate supersampling, multisampling, and compressed multisampling.

When using compressed multisampling, our system lets the user specify maximum and minimum sampling rates independently for both geometry and depths. Given these sampling rates we obtain the size of the color and depth tables together with the number of subsamples per pixel. We can then compute the memory required for each antialiasing buffer entry. Using this modified version of Mesa, any OpenGL application can now use compressed multisampling.

We produced results using four different configurations: no antialiasing, 2×2 multisampling, 4×4 multisampling, and our 4×4 compressed multisampling. For the compressed multisampling configuration we kept 3 colors and depths per table. Therefore, according to Equation 1, each antialiasing buffer entry requires 28 bytes.

4.1 Quality

In Figure 5, a horizontal grid of squares is rendered using the four different techniques. In these images, each square is composed of two triangles, creating a geometry discontinuity between each of them.

It can be seen that, for nearly horizontal and vertical lines, the 4×4 compressed multisampling scheme gives better results than standard 2×2 multisampling. This increase in image quality is even more noticeable in animated sequences.

In all the scenes we have rendered, 4×4 compressed multisampling gives results very similar to 4×4 multisampling. For example, the lower half of the third and fourth images of Figure 5 are almost identical. Differences between compressed multisampling and standard 4×4 multisampling appear near the horizon, where the

Scene	Resolution	Average Valid Subsamples per Pixel	Pixels with Number of Valid Subsamples				
			1–4	5–8	9–12	13–15	16
Checker	200×280	15.85	79	908	46	97	54870
Fan	300×300	15.99	0	3	6	6	89985
Teapot	200×300	15.93	36	149	219	323	59270
Gears	640×480	15.91	103	605	1985	4815	299692
Springs	151×115	15.89	9	49	141	268	16898
Plant	640×448	15.27	5814	7058	9101	8303	256444

Table 1: Quality measures for various scenes.

number of fragments per pixel causes overflow in the color and depth tables.

An example of bounded quality degradation can be seen in Figure 7. The branches of the plant in the left image were modeled using small triangles, resulting in a very large number of fragments and in table overflow. This can be seen in the top right image that shows with bright colors the pixels having many invalid subsamples. Leaves usually have more valid subsamples, thus appearing darker or black.

We know that a pixel with more invalid subsamples will result in a locally lower sampling rate. This is the case on the edges of the branch shown in the lower right image, yet we can see that a reasonable level of quality is maintained.

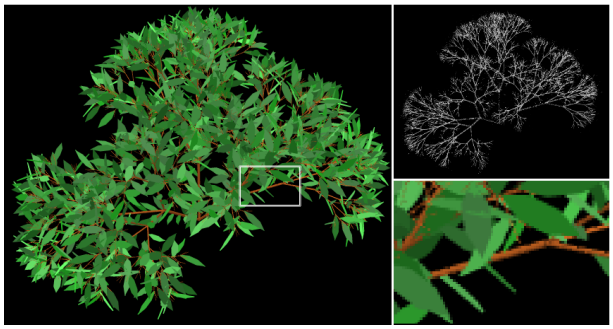

Figure 7: On the left, a complex plant rendered using 4×4 compressed multisampling. On the top right, image of the number of invalid subsamples for each pixel, dark pixels have less invalid subsamples than bright ones. On the bottom right, zoom on a branch of the plant.

We have evaluated the quality of our algorithm for the various scenes shown in Figure 9. For each of these we computed the average number of valid subsamples per pixel together with the distribution of the pixels having 1 to 16 valid subsamples. These results are detailed in Table 1.

4.2 Performance

For fixed color and depth table sizes, our compression and decompression algorithms can easily be unrolled and implemented in the hardware pipeline. Performance would then be limited mostly by the efficiency of data transfer to and from the antialiasing buffer.

We first compare the bandwidth required for fetching textures. Instead of evaluating the number of bytes read from texture memory, which depends on the type of filtering, we have directly counted the number of texture fetches. The results, for textured test scenes, are shown in Table 2. The number of required texture fetches do not change when using compression.

These results vary for each configuration even though multisampling performs a single texture fetch per fragment. This is due to the fact that the number of fragments generated by a single triangle increases with the number of subsamples per pixel. Table 2 therefore gives a good indication of the increase in the number of fragments processed by the pipeline when using multisampling. However, the bandwidth required for these extra texture fetches is much less important than the bandwidth due to exchanges with the antialiasing buffer.

Scene	No AA	2×2	4×4
Checker	16719	19997	22868
Fan	32827	34664	35469
Teapot	26239	32915	37331

Table 2: Number of texture fetches.

The most important increase in bandwidth is due to the data transfer with the antialiasing buffer. Each fragment going through the pipeline has to be compared with as many depths as there are subsamples in a pixel. When using standard multisampling, all these depths must be read from the antialiasing buffer.

When using compressed multisampling, we must read the set of indices together with the table of depths. However, the number of bytes transferred in this fashion will

usually be less than that of standard multisampling. This is due to the fact that the size of the tables is less than the number of subsamples per pixel.

When alpha blending is enabled the color of each subsample must also be obtained in order to compute the final color. In this case, compression allows us to transfer only the colors in the table, which is less than one color per subsample as required by standard multisampling.

Each fragment that passes through the pipeline without being culled must be written to the antialiasing buffer. In standard multisampling, this means transferring as many depth and colors as there are subsamples in a pixel. With compression, we must write back the antialiasing buffer entry.

The number of bytes exchanged with the antialiasing buffer is shown in Table 3. Notice that the number of bytes exchanged for 4×4 compressed multisampling is very similar to 2×2 multisampling.

Scene	No AA	2×2	4×4 compressed	4×4
Checker	0.38	1.83	1.75	8.40
Fan	0.75	3.17	2.72	13.08
Teapot	0.32	1.64	1.59	7.47
Gears	3.80	33.15	31.04	148.70
Springs	0.18	0.86	0.79	3.79
Plant	1.76	23.43	28.41	138.73

Table 3: Megabytes exchanged with the antialiasing buffer.

As we have seen, the visual results obtained with our 4×4 compressed multisampling are about as good as 4×4 multisampling. However, from Tables 2 and 3 we can see that it requires about the same bandwidth as 2×2 multisampling. From Section 4.1 and Table 3 we conclude that, in the 4×4 case, compression reduces bandwidth by about 80% without compromising quality.

5 Conclusion and Future Work

We have extended an edge antialiasing technique based on multisampling to handle compression of the antialiasing buffer. This technique can be implemented in hardware and achieves higher sampling rates while minimizing bandwidth requirements. It supports all the usual OpenGL fragment-related functions and is compatible with pixel shaders. As opposed to some previous antialiasing techniques, our compression scheme ensures that occluded subsamples will never contribute to the final image. Also, we can evaluate the minimal number of subsamples per pixel required by a given configuration of our system, leading to a lower bound on image quality.

In the future, we would like to evaluate performance and visual quality of compression when combined with low-cost techniques [3, 5]. To do so, we could extend the antialiasing buffer entries so that they contain subsamples coming from more than one pixel, as shown in Figure 8. We would also like to optimize memory organization in order to maximize cache efficiency and reduce even more the required data transfer. We could also antialias interpenetrating polygons through the use of low precision Z gradients [9]. Finally, we would want to study the impact of merging subpixels that share similar colors and depths.

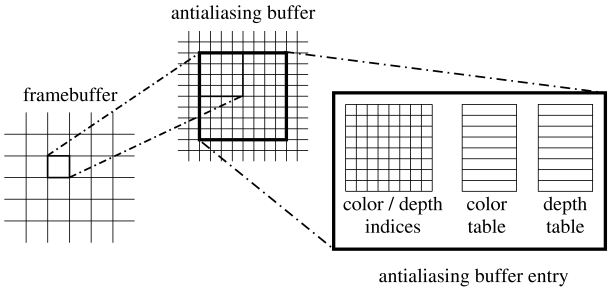

Figure 8: Extended structure of a single antialiasing buffer entry for a group of 16 subsamples shared by 4 neighboring pixels.

Acknowledgements

We would like to thank Jean-Jacques Ostiguy, at Matrox Graphics, for providing ideas and support when developing early versions of this work. We acknowledge financial support from NSERC.

References

[1] T. Aila, V. Miettinen, and P. Nordlund. Delay streams for graphics hardware. In *Proc. SIGGRAPH 2003*, pages 792–800, 2003.

[2] K. Akeley. RealityEngine graphics. In *Proc. SIGGRAPH 1993*, pages 109–116, 1993.

[3] T. Akenine-Möller. An extremely inexpensive multisampling scheme. Technical Report 03-14, Ericsson Mobile Platform AB, February 2003.

[4] T. Akenine-Möller and E. Haines. *Real-Time Rendering*. AK Peters, second edition, 2002.

[5] T. Akenine-Möller and J. Ström. Graphics for the masses: A hardware rasterization architecture for mobile phones. In *Proc. SIGGRAPH 2003*, pages 801–808, 2003.

[6] L. Carpenter. The A-buffer, an antialiased hidden surface method. In *Proc. SIGGRAPH 1984*, pages 103–108, 1984.

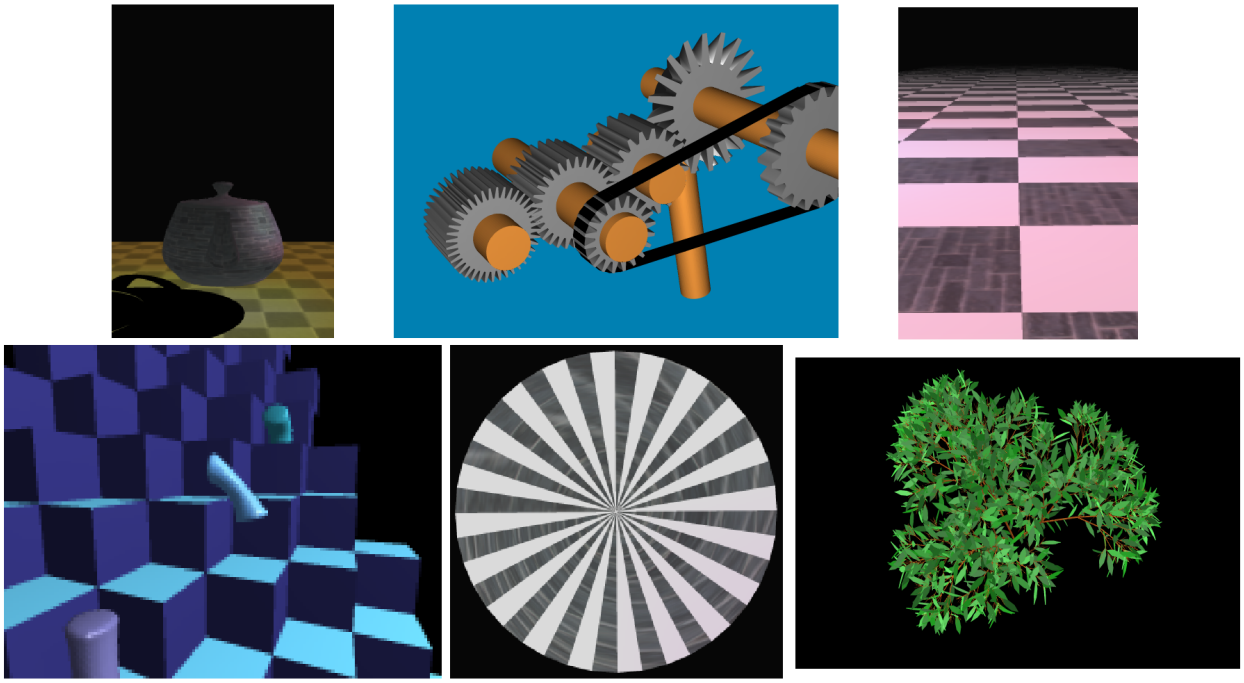

Figure 9: Results for various scenes rendered using 4×4 compressed multisampling. From left to right, top to bottom: Teapot, Gears, Checker, Springs, Fan, Plant.

[7] M. Deering and D. Naegle. The SAGE graphics architecture. In *Proc. SIGGRAPH 2002*, pages 683–692, 2002.

[8] P.E. Haeberli and K. Akeley. The accumulation buffer: Hardware support for high-quality rendering. In *Proc. SIGGRAPH 1990*, pages 309–318, 1990.

[9] N.P. Jouppi and C.-F. Chang. Z^3: An economical hardware technique for high-quality antialiasing and transparency. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 85–93, August 1999.

[10] A. Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics and Applications*, 9(4):43–55, 1989.

[11] Matrox. 16x fragment antialiasing. Technical report, 2002.

[12] J.S. Montrym, D.R. Baum, D.L. Digham, and C.J. Migdal. InfiniteReality: A real-time graphics system. In *Proc. SIGGRAPH 1997*, pages 293–302, 1997.

[13] NVIDIA. HRAA: High-resolution antialiasing through multisampling. Technical report, 2001.

[14] A. Schilling. A new simple and efficient antialiasing with subpixel masks. In *Proc. SIGGRAPH 1991*, pages 133–141, 1991.

[15] A. Schilling and W. Straßer. EXACT: Algorithm and hardware architecture for an improved A-buffer. In *Proc. SIGGRAPH 1993*, pages 85–92, 1993.

[16] P. Shirley. *Physically Based Lighting Calculations for Computer Graphics*. PhD thesis, University of Illinois at Urbana Champaign, 1990.

[17] L. Williams. Pyramidal parametrics. In *Proc. SIGGRAPH 1983*, pages 1–11, 1983.

[18] S. Winner, M. Kelly, B. Pease, B. Rivard, and A. Yen. Hardware accelerated rendering of antialiasing using a modified A-buffer algorithm. In *Proc. SIGGRAPH 1997*, pages 307–316, 1997.

[19] C.M. Wittenbrink. R-buffer: A pointerless A-buffer hardware architecture. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 73–80, 2001.

[20] www.mesa3d.org.