Z^3: An Economical Hardware Technique for High-Quality Antialiasing and Transparency

Norman P. Jouppi* — Compaq Chun-Fa Chang† — UNC Chapel Hill

Abstract

In this paper we present an algorithm for low-cost hardware antialiasing and transparency. This technique keeps a central Z value along with 8-bit floating-point Z gradients in the X and Y dimensions for each fragment within a pixel (hence the name Z^3). It uses a small fixed amount of storage per pixel. If there are more fragments generated for a pixel than the space available, it merges only as many fragments as necessary in order to fit in the available per-pixel memory. The merging occurs on those fragments having the closest Z values. This combines different fragments from the same surface, resulting in both storage and processing efficiency.

When operating with opaque surfaces, Z^3 can provide superior image quality over sparse supersampling methods that use eight samples per pixel while using storage for only three fragments. Z^3 also makes the use of large numbers of samples (e.g., 16) feasible in inexpensive hardware, enabling higher quality images. It is simple to implement because it uses a small fixed number of fragments per pixel.

Z^3 can also provide order-independent transparency even if many transparent surfaces are present. Moreover, unlike the original A-buffer algorithm it correctly antialiases interpenetrating transparent surfaces because it has three-dimensional Z information within each pixel.

1 Introduction

Aliasing is caused by insufficient sampling [2, 7]. To attenuate aliasing problems, the scene must be sampled at many positions within each pixel when it is rendered. Sampling can be done either uniformly or nonuniformly [9]. Nonuniform sampling methods, such as the stochastic sampling [6], are mostly implemented in software. Uniform sampling is also known as supersampling, and is implemented in most of the high-end graphics architectures today [1][20].

*Western Research Laboratory, Compaq Computer Corporation, 250 University Ave, Palo Alto, CA 94301 USA.
jouppi@pa.dec.com, http://www.research.digital.com/wrl/people/jouppi/bio.html

†Department of Computer Science, University of North Carolina at Chapel Hill, CB #3175, Sitterson Hall, Chapel Hill, NC 27599-3175 USA.
chang@cs.unc.edu, http://www.cs.unc.edu/ ∼ chang

A main issue of supersampling is the enormous amount of memory it requires. For example, a conventional 1280×1024 frame buffer with 32-bit color and 32-bit depth takes 10 Megabytes (MB) of memory. But with 4×4 supersampling, it requires more than 160 MB of memory. Worse than the memory capacity, 4×4 supersampling would require over 16 times the memory bandwidth, or for a given memory bandwidth slow down rendering by a factor of around 16.

Careful examination of a supersampled pixel with large numbers of sample points reveals that most of the color and depth values within a pixel are similar to values at other sample points. For some pixels partially covered by a foreground object and background objects, the color and Z values are clustered into groups with similar values. For pixels covered by objects steeply receeding from the viewer, although the subsample Z values may vary significantly, they still can be represented more compactly than many discrete values since they are a planar surface. What we need instead of brute-force supersampling is a way of taking advantage of these redundancies to compress the color and Z information into a smaller memory footprint, resulting in reduced cost, lower bandwidth requirements, and potentially higher system performance. We would also like these algorithms to require only a single pass and to be compatible with conventional rendering systems.

In software implementations of antialiasing, the use of dynamic memory allocation can be used to vary the amount of storage used by each pixel. However dynamic storage allocation is quite difficult and expensive to implement in hardware, so we would also like a compression technique that uses the same amount of storage per pixel. Given these requirements, such a compression technique would be expected to result in modest errors for more complex pixels (those with multiple surfaces of different colors) as compared to simple pixels which should be rendered exactly. This paper investigates techniques to make the resulting errors as small as possible (when judged by the human eye) while using the smallest amount of per-pixel memory.

1.1 Accurate Subpixel Z Values

Accurate treatment of subpixel Z values is in some ways more important than the accuracy of subpixel color values because small errors in Z values can lead to dramatically different pixel colors due to errors in occlusion calculations. Moreover, most real-world models have interpenetrating objects and fragments with overlapping Z ranges. Any technique that tries to reduce the storage required by Z entries has to pay special attention to various cases of interpenetrating and adjacent objects.

There are several posiblities for a more compact subpixel Z representation:

1. **Single Z at pixel center** — This has the advantage of simplicity, but provides the least information. Like other approaches that rely on one value, it is impossible to antialias interpenetrating surfaces based on a single value. Even worse, for fragments that do not cover the pixel center, the Z value associated with the fragment can be totally outside of its Z range. In the upper left example of Figure 1, this will lead

to fragment B being visible and fragment A not being visible, even though the reverse is true.

2. **Zmin and Zmax** — In the original A-buffer paper [4] a Zmax and a Zmin are used. These are used to estimate blending assuming the surfaces' slopes have opposite signs and the surfaces are interpenetrating. However, this case cannot be distinguished from the upper right case of Figure 1, since no information about the slopes are known. In this example, A and B should not be blended roughly equally, but instead fragment A completely obscures fragment B.

3. **Fragment subpixel Z average, or Centroid adjust** — One way to improve the accuracy in cases like the upper left of Figure 1 is to define each fragment's Z value to be the average Z value (or centroid) of the sample points covered by the fragment. This works in cases like the upper left example, but it still fails in others. Also, because it does not have any slope information, cases like the bottom left in Figure 1 will still not antialias. Instead the pixel will snap from fragment A's color to fragment B's color as B moves toward the viewer.

4. **Zdx and Zdy slopes** — As can be seen by the previous approaches, having complete subpixel Z information is crucial to proper rendering of many subpixel situations [14]. X and Y slope information in combination with Z specified at the pixel center can be used to regenerate individual subpixel Z values accurately[17]. Alternately, the EXACT method [19] computes the line of intersection between two fragment planes with Z slopes for both fragments using tables. This is input to methods which compute the pixel color based on area weighting [18]. In this paper we propose slope-based subpixel sampling techniques which we believe lead to easy implementation in VLSI designs.

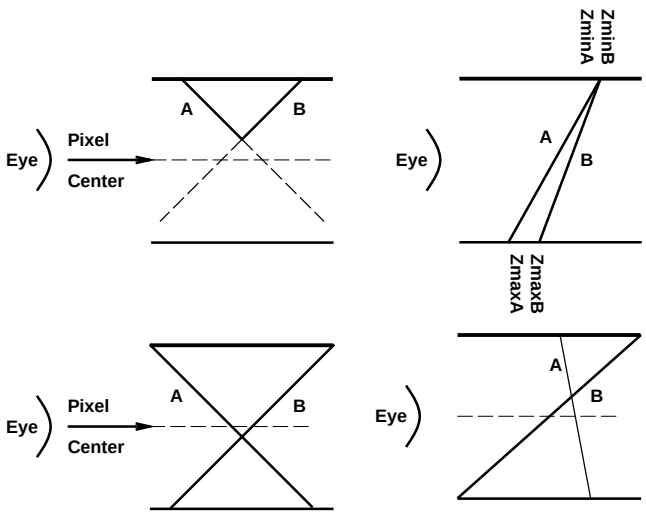

Figure 1: Difficult cases for non-complete Z information.

1.2 Order-Independent Transparency

Traditional implementations of supersampling do not support transparency unless objects are sorted before rendering. Even with triangle sorting, interpenetrating transparent fragments are not handled correctly. The A-buffer algorithm can provide the benefits of antialiasing and order-independent transparency at the same time, but does not correctly handle interpentrating opaque or transparent surfaces correctly. It also requires that all polygon fragments that can affect a pixel's color be kept until the drawing of the frame is complete. Only polygon fragments that are completely occluded by an opaque fragment may be deleted. Thus, in the worst case with many transparent objects, the A-buffer algorithm may require a potentially unbounded amount of memory for each pixel.

Two high-end graphics systems, the Megatek Discovery system[13] and the Sogitec AZtec system[5], both implement versions of the A-buffer algorithm. Neither of these systems use Z gradients. The Megatek Discovery system maintains fragment lists of up to 23 fragments per pixel[16]. In practice the Megatek implementation never merges fragments. Fragments can fall off fragment lists if they exceed the maximum list length. The Sogitec AZtec system merges fragments if they have the same object tag, their Z values differ by at most a predetermined value, they are non-overlapping, and they have colors that differ by at most a predetermined value. It also merges the last two fragments in a list if it runs out of per-pixel storage.

1.3 Our Algorithms

In this paper, we present the Z^3 algorithm for low-cost hardware antialiasing and order-independent transparency. It groups subpixels into fragments containing X and Y Z slopes plus center referenced Z values. Each slope is a one byte floating-point value, so this method has smaller memory requirements than sparse supersampling. It uses a small fixed amount of memory per pixel but a large number of sample points stored in a coverage mask. If the visible complexity of the pixel exceeds the storage space available for the pixel, the minimum number of fragments having the closest Z values are merged. This combines fragments from the same surface without leading to artifacts.

Z^3 can provide superior image quality to sparse supersampling methods that use eight samples per pixel while using storage for only three fragments. This technique also makes the use of large numbers of samples (e.g., 16) feasible in inexpensive hardware. It is simple to implement because it uses a small fixed number of samples per pixel. Like traditional supersampling techniques it properly antialiases opaque interpenetrating objects. However, it also provides order-independent transparency and antialiasing of interpenetrating transparent objects. Z^3 can provide order-independent transparency even if many transparent surfaces are present, albeit at a cost of slightly more memory.

2 Related Work

In the traditional graphics pipeline [8], the polygons which describe the surface of objects to be rendered are rasterized into a frame buffer and a depth buffer.

2.1 Supersampling and A-buffer Techniques

Both supersampling and the A-buffer consume too much memory in their original forms for low-cost implementations. Several multi-pass rendering algorithms have been proposed to reduce the memory requirements of supersampling and the A-buffer. The Accumulation buffer [10] can produce the quality of supersampling without increasing the size of frame buffer. The Virtual Pixel Maps technique proposed by Mammen [11] and its variation proposed by Winner et. al. [21] replace A-buffer's unbounded list of visible objects by a moving depth buffer, in addition to the Z buffer used by opaque surfaces. However, the performance of these algorithms suffer due to their multi-pass nature.

2.2 Subpixel Sampling Methods

There are several different ways to sample points within a pixel. Software methods[6] may use many sample points (e.g., 100) with near random distributions, but hardware is typically limited to a modest number of sample points (16 or less) on a regular grid. Early hardware approaches such as the Silicon Graphics (SGI) RealityEngine [1] used a 4×4 array of sample points. A simple 4×4 array of points has the disadvantage of producing only a few intensity steps for moving edges that are near vertical or horizontal. We call this approach *full supersampling*.

To address this limitation, Schilling [18] proposed an area weighting method which can give a full range of intensities as a near vertical or horizontal edge is moved across a pixel. Near vertical or horizontal edges are important because they can produce "jaggies" that turn into distracting "crawlies" when animated. However, this method has the disadvantage of lighting subsamples which are not actually covered by the primitive, and so can lead to artifacts. A variant of this technique was implemented in SGI's RealityEngine.

More recently, a technique we call *sparse supersampling* has appeared in SGI's Infinite Reality [15]. Here the number of actual sample points is less than the number of potential sample points in the grid. By chosing at most one sample point on each row and column, it is possible to get n intensity steps from n sample points distributed on a $n \times n$ grid for moving near vertical or horizontal edges. It is also possible to choose the sample points so that each quadrant of the pixel has similar weighting. This is important to prevent flashing of sub-pixel sized moving objects.

Sparse supersampling can give more accuracy than full supersampling for a given number of sample points without introducing artifacts. Therefore in most of our work we use sparse supersampling.

3 Test Datasets

For testing our algorithms with opaque surfaces, we use a model of an x-wing fighter (see Figure 8). Each of the 6084 triangles in the model is a random flat-shaded color. A realistically shaded fighter would have noticeable aliasing mainly at its silhouette. By assigning each triangle in the model a random color we create noticeable aliasing artifacts at each triangle edge. Furthermore, we disabled backface culling to increase the depth complexity of the image, and hence the number of fragments processed in each pixel. Figure 2 shows the maximum number of opaque surfaces that appear within each pixel during the rendering of a typical scene when there are 16 samples per pixel. Note that on the gun turrets and some areas of the engines and fuselage there are pixels that require almost as many fragments as sample points for fully accurate rendering. Finally, we render the image at a small scale to create many subpixel-sized features.

The x-wing fighter model has too great a depth complexity to make a practical test of transparency. For testing transparency algorithms, we render a Cessna seaplane consisting of 2239 transparent triangles. Figure 10 shows an image of the Cessna. The maximum number of transparent surfaces that appear within each pixel during the rendering of the Cessna when there are 16 samples per pixel is given in Figure 3. Even though this image assumes 16 sample points, some pixels require as many as 30 fragments because of the depth complexity of the transparent fragments.

To test interpenetrating surfaces we also have a model of a beach ball with transparent and opaque stripes penetrating a checkerboard. This test case only appears in the video accompanying this paper.

The test images in this paper are all at a resolution of 128×96 pixels. This resolution provides a resolution similar to a 19" 640x480 monitor so that individual pixels can be distinguished on the paper. The video consists of two side by side 180x240 images or a single 360x240 image, unless stated otherwise. This uses the maximum vertical resolution available in one field of NTSC video.

Supersampling objects containing subpixel-resolution lines using small numbers of sample points can lead to "Marquee light" artifacts, similar to a line of moving theater lights. To test the performance of the algorithms in these conditions the x-wing fighter and Cessna test cases both have subpixel width lines. The x-wing fighter has a yellow line down the port side of its fuselage, and the Cessna has a green line between its wings and its ailerons and flaps. Please note these during the video.

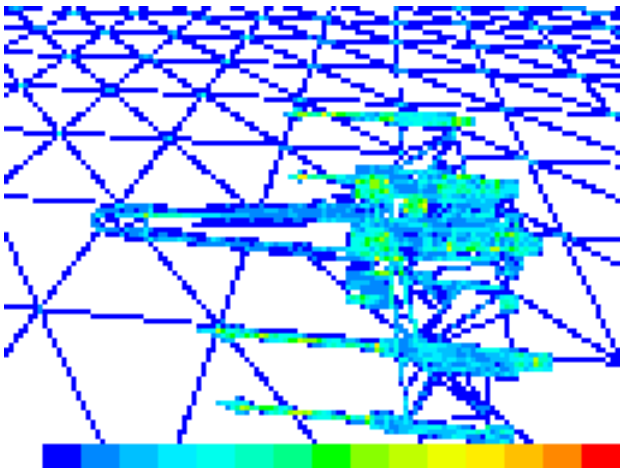

Figure 2: The color represents the maximum number of fragments that are visible in each pixel of one frame of the x-wing test case when using 16 samples per pixel. The color code at the bottom shows the colors representing 1 (white) to 16 (red). (Also in the color section.)

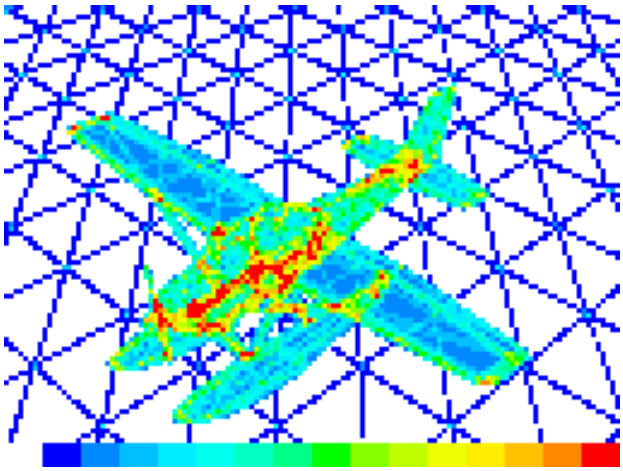

Figure 3: The color represents the maximum number of fragments that are visible in each pixel of one frame of the transparent Cessna test case when using 16 samples per pixel. The color code at the bottom shows the colors representing 1 (white) to 16 or more (red). Some pixels have as many as 30 fragments. (Also in the color section.)

The error metric we use throughout the paper is the sum of the squares of the per pixel color difference:

$$\text{error} = \sum_{\forall i,j} \sum_{c=R,G,B} ((p_{ijc} - q_{ijc})^2)$$

where p_{ij} and q_{ij} are pixels from the same location of a test image and a reference image. The RGB components of the pixel color are within the range of 0 to 255. The square of the error is chosen because a small number of pixels with large errors are more noticeable than a large number of pixels with small errors. We have also evaluated the maximum error. This behaves similarly to the sum of the squared per pixel errors, but is less representative of the image as a whole.

4 The Z^3 Algorithm

Figure 4 describes the pixel data structure that is used in our algorithm. Rather than providing a separate color, Z, and stencil (fragment collectively) for each sample point, we only provide a few fragment entries per pixel. Each fragment entry has a m-bit coverage mask that indicates which of the m sample points in the pixel are covered by the fragment. Fragment color values are the average of the color values at the covered sample points. Z values are specified at the center of the pixel, but compact X and Y Z gradients are also kept. While a pixel is updated and while the DRAM page is still open the final pixel color is computed and stored in memory.

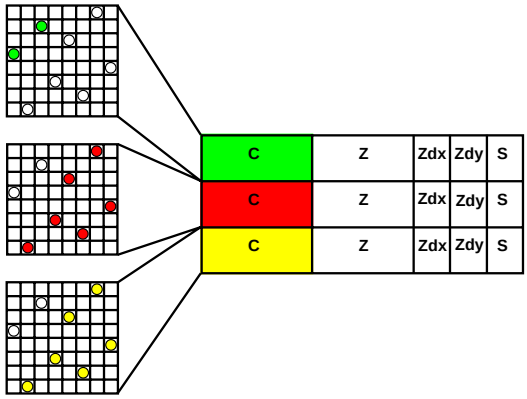

Figure 4: The Z^3 data structure, which is used to provide antialiasing and order-independent transparency. In this example each of the three fragments has an associated coverage mask, which contains 8 samples on an 8×8 grid. Implementations vary in the number of sample points and fragments. This data structure is used in addition to coventional storage of per-pixel final pixel color and Z information in front and back buffers.

There are several parameters that determine the size of this data structure:

1. m: m sample points are used per pixel.
2. k: k fragments per pixel are kept.
3. c, z, zdx, zdy, s: a c-bit color, z-bit depth, two g-bit z slopes stored in a floating-point format, and s-bit stencil are used.

The total size of each pixel is then:

$$k \times (m + c + z + 2g + s) \text{bits}.$$

The floating-point Z gradients do not need to be extremely accurate to result in correct results in most circumstances. For example, imagine we are using a 24-bit integer Z value. To capture the whole range of possible Z gradients, you would need a fixed-point Z gradient larger than 24 bits. But an 8-bit floating-point Z gradient can consist of a sign bit, 5 bit exponent, and 3 bit mantissa. (These 9 bits are stored in 8 bits utilizing a hidden msb mantissa bit as in the IEEE floating point standard, since the msb of a floating mantissa is always 1 unless the whole number is zero, which is denoted by a zero exponent.) The 5 bit exponent (ranging from 2^{28} to 2^{-3}) can cover the entire range of the 24-bit fixed point Z value plus additional fractional values. Three bits of mantissa provides more than enough precision in the vast majority of cases where the Z gradient is needed.

If the per fragment Z-value is also stored as a floating point format, it is usually done to represent a Z value with a slightly larger range in a more compact format. In this case a slope exponent is more likely to be about 6 bits, and the Z range could also include some fractional values by using a biased exponent. This still leaves 2 bits for the mantissa and one for the sign.

Because the floating-point slopes have such small mantissas, they can easily be converted to fixed point Z slopes by small 2 or 3 bit wide shifters. Since there are a small number of sample points (e.g., 16 or less) on a small regular grid which is a power of two (e.g., 16x16), the calculation of the actual sample point Z values from the slopes involves multiplication of each Z gradient by a small offset that specifies the distance from the center (e.g., a fraction less than 1/2 such as 5/16 or 2/16). The division by a power of two (e.g., 16) is a shift, while the multiplication by a small constant (e.g., 5 or 2) can be performed by at most a few levels of carry-save adders. The result of these for X and Y go into carry-save adders along with the center-referenced Z value and then on to a carry-lookahead adder to calculate the actual Z value at that sample point. The total complexity of this is similar to multiplying the center-referenced Z values by a small number at each sample point. This requires much less hardware than storing the Z value and providing adequate read/write bandwidth for the tens of millions of sample points on a screen.

4.1 Overview of Fragment Processing

Unfortunately, there are not always a small fixed number of visible fragments per pixel, and in some cases we will need more fragment entries than we have storage locations. This is particularly true when we have a relatively large number of sample points in comparison to the number of available fragments, or when transparent objects are being rendered. In the worst opaque case each of the eight sample points in Figure 4 might be on a different fragment. If we only had storage for three fragments, we would have almost three times more information than we had space for. In the worst transparent case, the visible transparent depth complexity is virtually unbounded. Each transparent surface could also be fractured into many subpixel-sized fragments.

In general, if we have more fragments than we have locations for fragment storage, some information will be lost and this can lead to artifacts. The algorithms we have developed attempt to minimize the information lost as well as the possible artifacts produced. The algorithms are complicated by the fact that they must make decisions as the scene is being rendered without any information about what future rendering operations may do.

The basis of our algorithms involves merging fragments that are very close in their Z values. This combines fragments that are part of the same surface, but have been broken into multiple fragments by tesselation. We can also combine two transparent surfaces that are very close in Z value. This reduces the visible transparent depth complexity and in most cases results in no difference in pixel color.

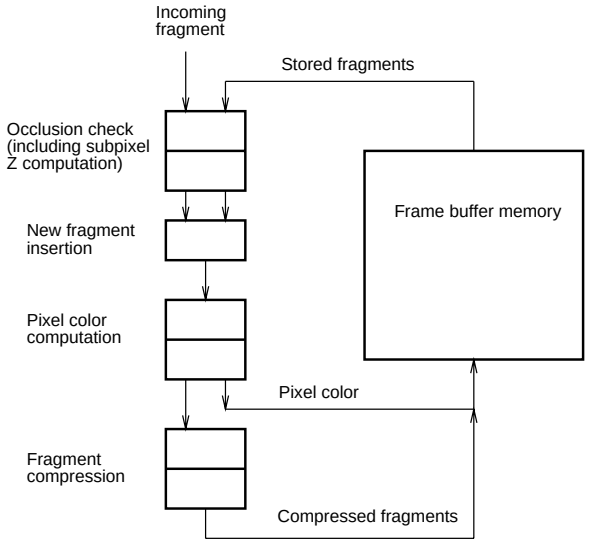

Figure 5: The Z^3 fragment processing pipeline.

There are four main steps that are taken when a new fragment arrives at a pixel. The steps form a fragment processing pipeline (see Figure 5). Existing fragments are stored in frame buffer memory sorted based on their center Z value. When a new fragment arrives, the existing fragments are read in starting with the closest fragment. The four new fragment processing steps are:

1. **Occlusion Check** — the sample points that are covered by the new fragment are checked whether they occlude or are occluded by any stored fragments. This is done by comparing the depth value recomputed for each sample point from the center Z value and the X and Y Z slopes. If a stored fragment is completely occluded by the new fragment, its storage can be freed for later use.

2. **Fragment Insertion** — If any sample points of the new fragment pass the occlusion test, the new fragment is inserted in the pipeline of existing fragments in the proper place based on its center weighted Z value. This can be done by comparing the new fragment's center referenced Z value with two adjacent stages in the fragment pipeline. If the new fragment Z value is larger than the first stage but less than the second stage, when the pipeline is shifted next the new fragment is loaded into the first stage while the second fragment and those behind it do not advance.

3. **Pixel Color Computation** — The pixel color is computed before any compression required by the addition of the new fragment. Thus the pixel color is based on all the information in the existing fragments and the new fragment. Details of the pixel color computation including computation of the swap vector are described below in Section 4.2.

4. **Fragment Compression** — If there are more fragments than storage locations, one fragment will need to be merged with another. This is described in more detail below in Section 4.3.

4.2 Pixel Color Computation

Because the fragments within a pixel are sorted in depth order, we can usually compute the color of each pixel by alpha blending whole fragments. A box filter is then applied to produce the final pixel color, although our algorithm is extendible to more complex filters [3].

Unfortunately when transparent fragments overlap in their Z ranges with other fragments (which may or may not be transparent), computing the final pixel color based on the sorting implied by the center-referenced Z values can create erroneous results. Consider the lower-right case in Figure 1. The transparent fragment A is actually partially in front of opaque fragment B, even though its center-referenced Z value is behind it. If A is processed first, the opaque fragment B will completely obscure fragment A instead of blending with the portion of A in front of fragment B.

Before computing the color at each sample point we compare the per-sample point Z values in adjacent stages of the pipeline. If their front-to-back order is wrong, we set a bit in a swap vector between the pipestages. The swap vector tells us if any sample points have their order reversed between pairs of fragments. Introduce a figure like the patent here.

In this way, we are guaranteed that we can correctly reorder all sample points where one fragment interpenetrates an adjacent fragment at that sample point. We do not correctly handle arbitrary interpenetration, such as one perpendicular fragment interpenetrating many parallel fragments. However, such cases are rare, and moreover the error in such cases is not large because of the many surfaces viewed in series and the small coverage of the perpendicular fragment. If we swapped the processing order of two fragments while computing the color of a subpixel, we save this bit of information in a swap vector for possible later use during fragment compression.

After the fragments are reordered for each sample point, we then compute the pixel color and alpha on a per-sample point basis. We sum the colors from all the sample points and divide by the number of samples per pixel (i.e., right shift).

4.3 Fragment Compression Algorithm

Fragment compression only takes place when the number of fragments exceeds the preset limit k. Because the fragments are sorted in order of increasing center Z values, we know that the two closest fragments (in terms of their center Z values) are adjacent to each other in the pipeline. Although differences between center Z values and per sample point Z values are significant for occlusion and color calculations, we have found that center Z values are adequate for merging of fragments. As the fragments pass through the pipeline, they pass by a subtractor which computes the difference in center Z values between the adjacent stages. Based on these results, one of the k adjacent pairs of fragments out of the $k + 1$ fragments are merged.

Because merging may introduce errors, we would like to minimize the extent of these errors. In general, changes to fragments covering a small number of sample points result in smaller pixel errors than changes to fragments covering a large number of sample points. Also, the information content (in terms of the final pixel color) of a fragment entry covering many sample points is higher than that of an entry covering just one or a few sample points. For this reason we also weight the Z difference calculations by the minimum of the sample coverage counts of the two fragments. What this does is bias the selection towards the combining of small fragments that may be a little further apart rather than larger fragments that may be a little closer. We have found that this improves the final image quality.

To handle merging of interpenetrating transparent fragments correctly, we use information that was saved during pixel color computation in a swap vector. There is a swap vector for each pair of fragments and it has a bit for each sample point. The bit is set when the order of a pair of fragments must be swapped during color computation of that sample point due to interpenetration. After the

swap vector has been computed on either side of a fragment we can throw away the per sample-point Z information to save circuitry.

In our algorithm, the center Z values of the two merging fragments are weighted averaged based on the number of sample points that they cover. Weighted averaging of gradients works in many situations, but does not work in situations where one of the fragments is being viewed edge-on such as the side of a cylinder. These fragments may have extremely large gradients (approaching the maximum Z value) that will still be extremely large after averaging but cover much more of the pixel. Instead for each of the incoming fragments, we compute the absolute value of the X and Y gradients (by setting the sign bit to zero). We set the merged fragment's X and Y gradients to those X and Y gradients with the smallest magnitude. The stencil of the fragment covering the most samples is copied to the combined fragment.

The merging of the adjacent fragment pair is complicated by transparency. If both fragments are opaque, their color contents could be simply combined with weighted averaging based on the number of sample points each one covers. If one or both of the fragments are transparent, the calculation of pixel color must be done similarly to the final pixel color computation in section 4.2. However, in this case the merged fragment may not cover all the sample points in the pixel. To handle this properly, after the per sample colors have been computed and summed (including use of the swap vector to get the per sample point ordering correct), the result is multiplied by a fraction that is the percent of the pixel covered by the merged fragment.

The following equations describe the color computations performed at each sample point when merging two fragments, assuming 8-bit alpha and color channels. Sample points uncovered by either fragment return zero. Sample points covered by only one fragment return the alpha and each color channel multiplied by the alpha of that fragment. For sample points covered by both fragments, the following computations are made independently for each sample point using the swap vector to determine which fragment is in front and which is in back.

$$\alpha_{sample} = \alpha_{front} + \frac{(1 - \alpha_{front}) * \alpha_{back}}{255}$$

$$C_{sample} = C_{front} * \alpha_{front} + \frac{C_{back} * \alpha_{back} * (1 - \alpha_{front})}{255}$$

where C is each of the color channels R, G, and B. This computes the transparency and reflected light for each of the colors (multiplied by 255) for each sample point. Then the alphas and color channels from each sample point are summed. The number of sample points covered by the merged fragment cnt_m is computed by OR'ing together the two coverage masks and counting the population of 1's. Then the final merged fragment color and alpha is computed as follows:

$$\alpha_{merged} = \frac{\sum \alpha_{sample}}{cnt_m}$$

$$C_{merged} = \frac{\sum C_{sample}}{\alpha_{merged} * cnt_m}$$

for each of the color channels R, G, and B.

4.4 Performance on the Opaque Test Case

We use the terminology $N\times$ sparse supersampling to denote $N \times N$ sparse supersampling with N samples. Figure 6 shows the errors introduced by 3×3 and 4×4 full supersampling, 4×, 8×, and 16× sparse supersampling, and 4×, 8×, 16× and 32× sparse supersampled Z^3 with various numbers of fragments. The reference image for this figure is rendered by 16×16 full supersampling.

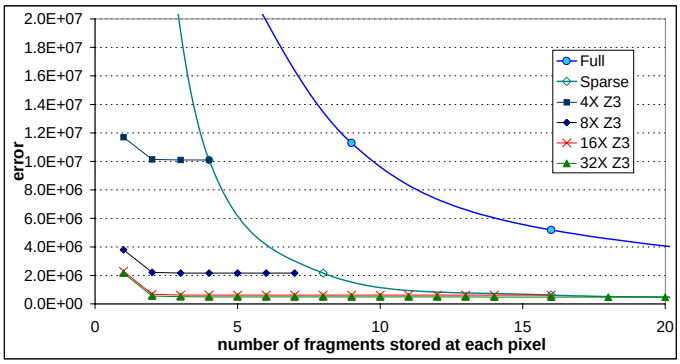

Figure 6: The errors introduced by 3×3 and 4×4 full supersampling, 4×, 8× and 16× sparse supersampling, and 4×, 8×, 16× and 32× sparse supersampled Z^3 with various numbers of fragments. The reference image is rendered by 16×16 supersampling.

Figure 6 shows two things. First, it is obvious that full supersampling consumes too many fragments to reduce error cost-effectively. For example, 4×4 supersampling takes 16 fragments and does not even perform as well as 8× sparse supersampling which takes only 8 fragments. Second, we can improve upon the performance of sparse supersampling by using sparse supersampled Z^3 with more samples and fewer fragments. For example, 16× sparse supersampled Z^3 with 3 fragments results in less than half the pixel error as 8× sparse supersampling with 8 fragments, while using only about half the storage.

Figure 7 is an image of the x-wing fighter rendered with sparse supersampling with 4 samples on an 4×4 grid. It is lower fidelity than Figure 8 which has storage for only 3 fragments but uses 16 samples on a 16×16 grid.

4.5 Performance on the Transparent Test Case

Figure 9 shows the error from 4×, 8×, and 16× sparsely supersampled Z^3 with various numbers of fragments for the transparent Cessna in comparison to 32× sparse supersampling with an unbounded number of fragments. Figure 10 shows the Cessna seaplane model rendered with storage for 4 fragments per pixel. Each pixel has 16 samples taken from an 16×16 grid. The lower part of the figure shows the difference between it and an image rendered with an unlimited number of fragments per pixel. Note that although some pixels have a depth complexity of 16 or more (as shown in Figure 3), using 4 fragments per pixel produces acceptable results. Most of the error is confined to areas that have 16 or more visible fragments per pixel, and this results in a small amount of “visible noise” in the image. Since the noise is relatively small but varies from frame to frame it is more visible in the video.

4.6 Implementation

The use of a small fixed number of fragment storage locations per pixel has the advantage of having low overhead and design simplicity. Although fragments in some pixels will be unused while other pixels could use additional fragment storage, the fixed allocation policy is easy to implement and verify. In practice, the errors introduced by having a modest fixed number of fragments

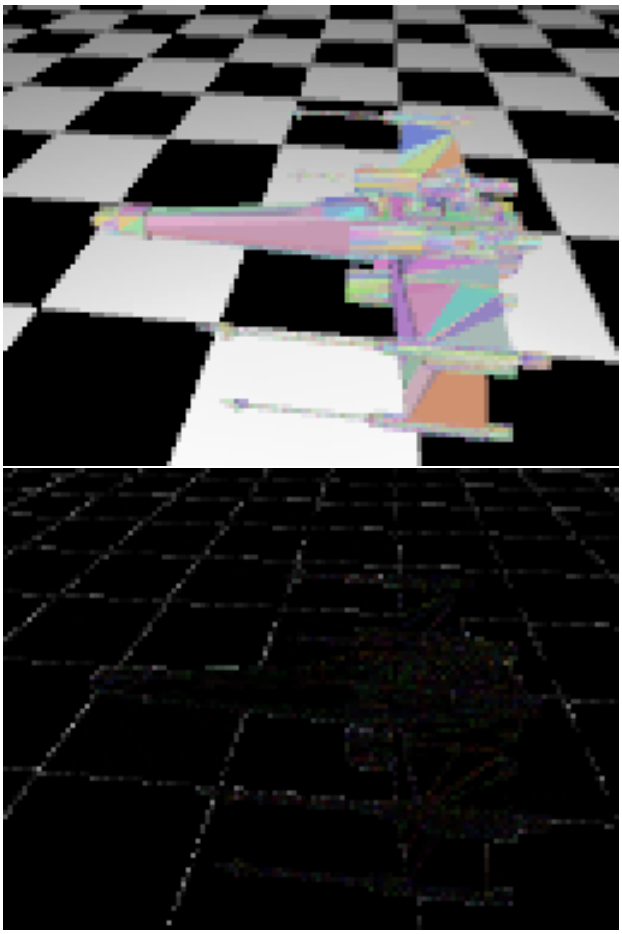

Figure 7: The x-wing fighter with traditional 4X sparse supersampling (using 4 samples on a 4x4 grid). The lower figure gives the per-pixel difference from a 16X sparse supersampled reference image.

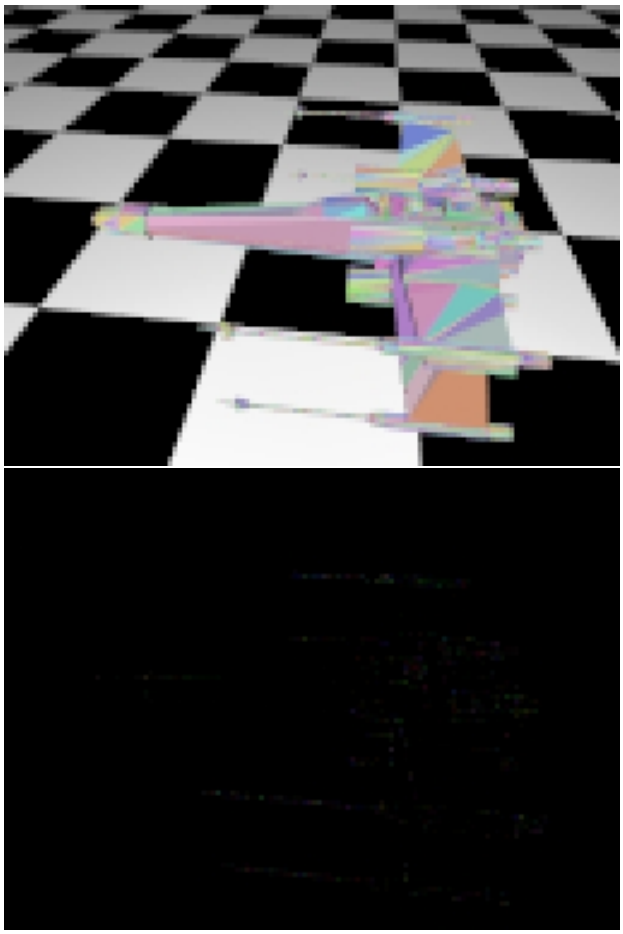

Figure 8: The x-wing fighter with 16 samples on a 16×16 grid and storage for 3 fragments. The lower figure gives the per-pixel difference from a 16X sparse supersampled reference image. (Also in the color section.)

per pixel appear small compared to losses in pixel information due to a limited number of sample points. For example, according to Figure 6, the error introduced by having storage for only 3 fragments per pixel is much smaller than the error from having only 8 or 16 sparse samples instead of the full 256 samples of the reference image.

Since Z^3 uses less memory per pixel than sparse supersampling with the same number of sample points, when a pixel is accessed less memory bandwidth will be necessary. This can either enable higher performance for a given system memory bandwidth, or lower the bandwidth requirements (and hence lower the cost) for a given level of performance.

To provide 16X sparse supersampled Z^3 with storage for 4 fragments per pixel requires about 50 bytes per pixel. Hence a 1280×1024 resolution screen would require about 64MB of frame buffer memory (not including textures), which at recent prices would cost about 80 dollars. Based on long-term historical DRAM price trends, this should reduce to about 20 dollars of additional memory expense in three years. After consultation with some of the designers of the Neon graphics accelerator[12], it appears that support for our algorithms will be well within the capabilities of next-generation ASIC fabrication processes.

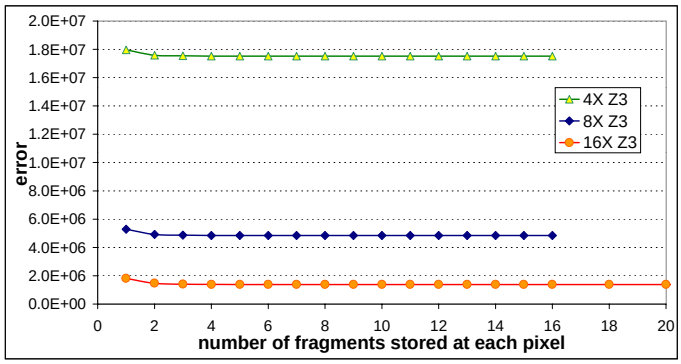

Figure 9: The errors introduced by 4×, 8×, and 16× Z^3 with various numbers of fragments. The reference image is rendered by 32× sparse supersampling with an unbounded number of fragments.

5 Conclusions

The Z^3 algorithm can provide economical high-quality hardware antialiasing and order-independent transparency. It uses a small

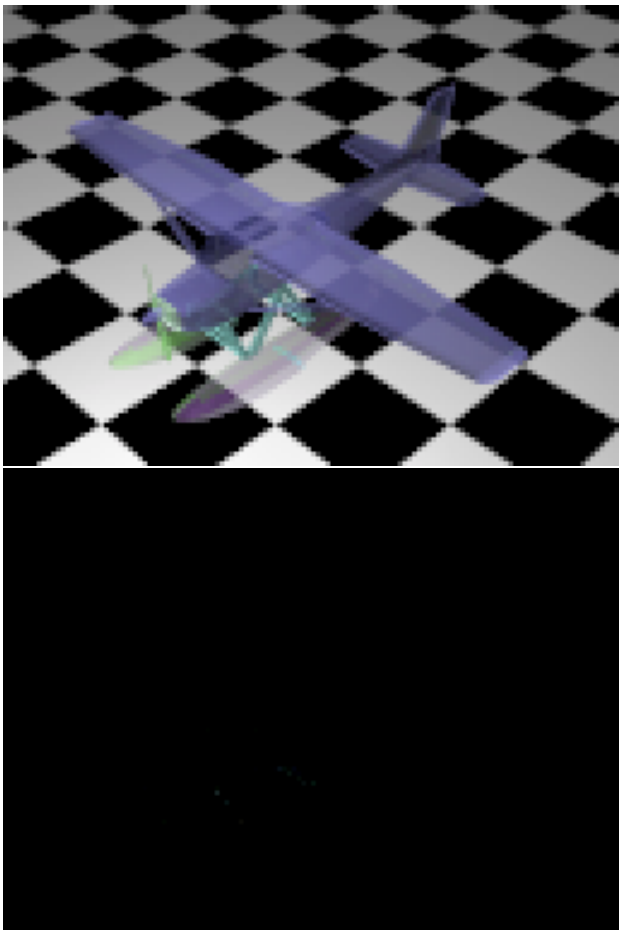

Figure 10: The transparent Cessna seaplane with 16 samples on a 16×16 grid and storage for 4 fragments per pixel. The lower figure gives the per-pixel difference from a 16X sparse supersampled reference image with an unbounded number of fragments. (Also in the color section.)

fixed amount of storage per pixel. If there are more fragments generated for a pixel than the space available, it merges only as many fragments as necessary in order to fit in the available per-pixel memory. The merging occurs on those fragments having the closest Z values. This combines different fragments from the same surface, resulting in both storage and processing efficiency.

When operating with opaque surfaces, Z^3 can provide superior image quality over sparse supersampling methods that use eight samples per pixel while using storage for only three fragments. Z^3 also makes the use of large numbers of samples (e.g., 16) feasible in inexpensive hardware, enabling higher quality images. It is simple to implement because it uses a small fixed number of fragments per pixel.

Z^3 can also provide order-independent transparency even if many transparent surfaces are present. Moreover, unlike the original A-buffer algorithm it correctly antialiases interpenetrating transparent surfaces because it has three-dimensional Z information within each pixel.

As memory prices continue to drop and VLSI integration increases, these techniques could enable high-quality antialiasing and transparency on all but the cheapest computer platforms.

6 Acknowledgements

Joel McCormack inspired the original work on this topic. Brian Pinz introduced the authors to A-buffer methods. Joel McCormack, Keith Farkas, Ron Perry, and Bart Sano commented on a early draft of this paper. Finally, the comments of the anonymous reviewers with constructive comments were much appreciated.

References

[1] Kurt Akeley. RealityEngine Graphics. In *Computer Graphics Annual Conference Series (Proceedings of SIGGRAPH 93)*, pages 109–116, August 1993.

[2] James F. Blinn. Jim Blinn's Corner: What We Need Around Here is More Aliasing. *IEEE Computer Graphics and Applications*, 9(1):75–79, January 1989.

[3] James F. Blinn. Return of the Jaggy. *IEEE Computer Graphics and Applications*, 9(2):82–89, March 1989.

[4] Loren Carpenter. The A-buffer, an Antialiased Hidden Surface Method. In *Computer Graphics Annual Conference Series (Proceedings of SIGGRAPH 84)*, volume 18, pages 103–108, July 1984.

[5] J. C. Chauvin. An Advanced Z-Buffer Technology. In *Proceedings of the IMAGE VII Conference*, pages 77–85, Tucson, June 1994.

[6] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed Ray Tracing. In *Computer Graphics Annual Conference Series (Proceedings of SIGGRAPH 84)*, volume 18, pages 137–45, July 1984.

[7] Franklin C. Crow. The Aliasing Problem in Computer Generated Shaded Images. *Communications of the ACM*, 20(11):799–805, November 1977.

[8] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1990.

[9] Andrew S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufmann, San Francisco, California, 1995.

[10] Paul E. Haeberli and Kurt Akeley. The Accumulation Buffer: Hardware Support for High-Quality Rendering. In *Computer Graphics Annual Conference Series (Proceedings of SIGGRAPH 90)*, volume 24, pages 309–318, August 1990.

[11] Abraham Mammen. Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique. *IEEE Computer Graphics and Applications*, 9(4):43–55, July 1989.

[12] Joel McCormack and et. al. Neon: A Single-Chip 3D Workstation Graphics Accelerator. In *Proceedings of the Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 123–132, Lisbon, Portugal, Sep 1998.

[13] Megatek. Instacuity White Paper. 1993.

[14] Steven E. Molnar. *Image-Composition Architectures for Real-Time Image Generation*. PhD thesis, University of North Carolina at Chapel Hill, 1991. Available as UNC-CH Computer Science TR91-046, at http://www.cs.unc.edu/Research/tech-reports.html.

[15] John Montrym, Daniel Baum, David Dignam, and Christopher Migdal. InfiniteReality: A Real-Time Graphics System. In *Computer Graphics Annual Conference Series (Proceedings of SIGGRAPH 97)*, pages 293–302, August 1997.

[16] Brian Pinz. Private communication. 1998.

[17] C. Romanova and U. Wagner. A VLSI Architecture for Antialiasing. In *Proceedings of the 4th Eurographics Workshop on Graphics Hardware*, 1989.

[18] Andreas Schilling. A New Simple and Efficient Anti-aliasing with Subpixel Masks. In *Computer Graphics Annual Conference Series (Proceedings of SIGGRAPH 91)*, volume 25, pages 133–141, July 1991.

[19] Andreas Schilling and Wolfgang Straßer. EXACT: Algorithm and Hardware Architecture for an Improved A-buffer. In *Computer Graphics Annual Conference Series (Proceedings of SIGGRAPH 93)*, volume 27, pages 85–92, August 1993.

[20] Jay Torborg and James T. Kajiya. Talisman: Commodity Realtime 3D Graphics for the PC. In *Computer Graphics Annual Conference Series (Proceedings of SIGGRAPH 96)*, pages 353–364, New Orleans, Louisiana, August 1996.

[21] Stephanie Winner, Mike Kelly, Brent Pease, Bill Rivard, and Alex Yen. Hardware Accelerated Rendering of Antialiasing Using a Modified A-buffer Algorithm. In *Computer Graphics Annual Conference Series (Proceedings of SIGGRAPH 97)*, pages 307–316, Los Angeles, California, August 1997.

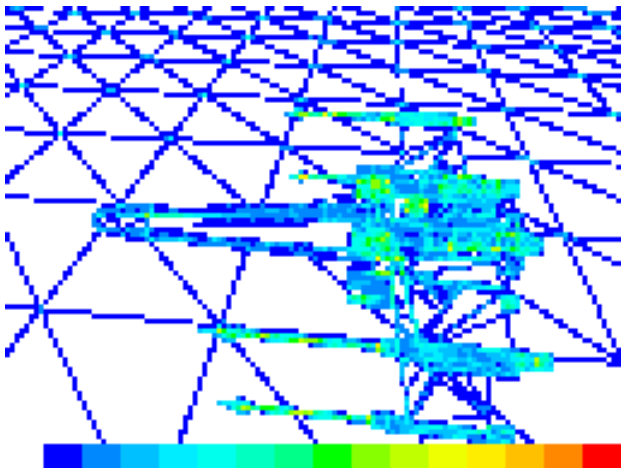

Figure 3: The color represents the maximum number of fragments that are visible in each pixel of one frame of the x-wing test case when using 16 samples per pixel. The color code at the bottom shows the colors representing 1 (white) to 16 (red).

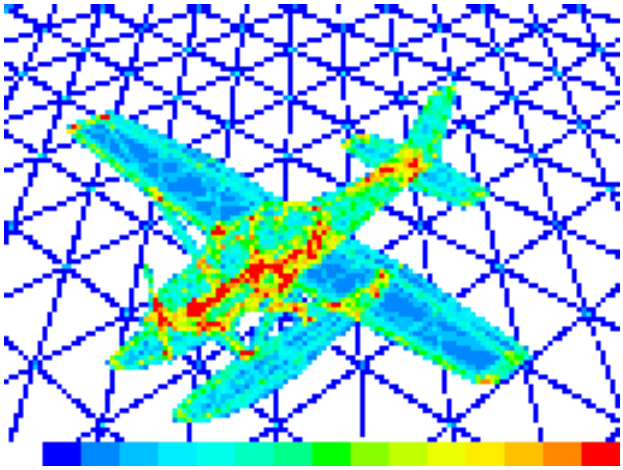

Figure 4: The color represents the maximum number of fragments that are visible in each pixel of one frame of the transparent Cessna test case when using 16 samples per pixel. The color code at the bottom shows the colors representing 1 (white) to 16 or more (red). Some pixels have as many as 30 fragments.

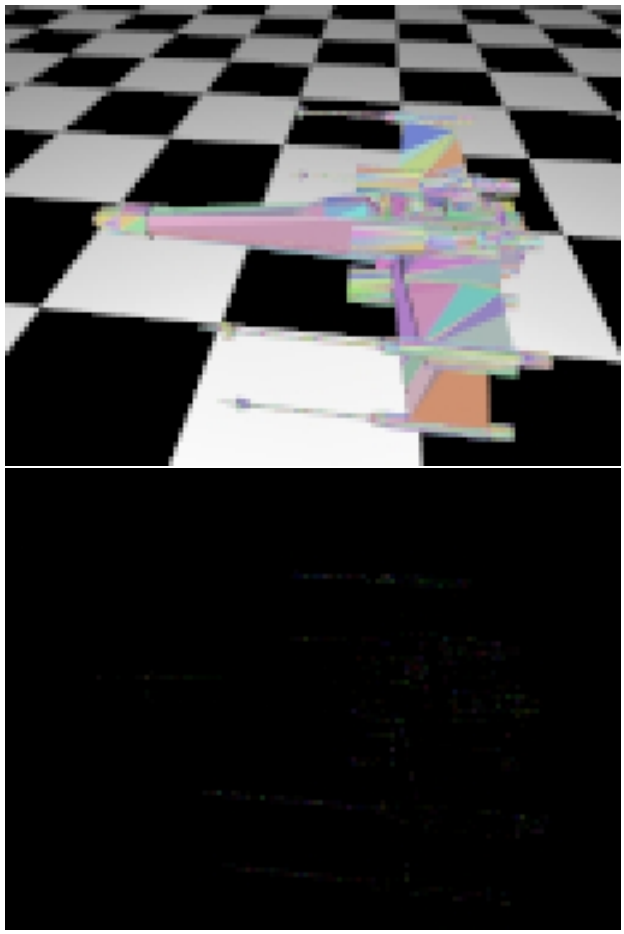

Figure 9: The x-wing fighter with 16 samples on a 16×16 grid and storage for 3 fragments. The lower figure gives the per-pixel difference from a 16X sparse supersampled reference image.

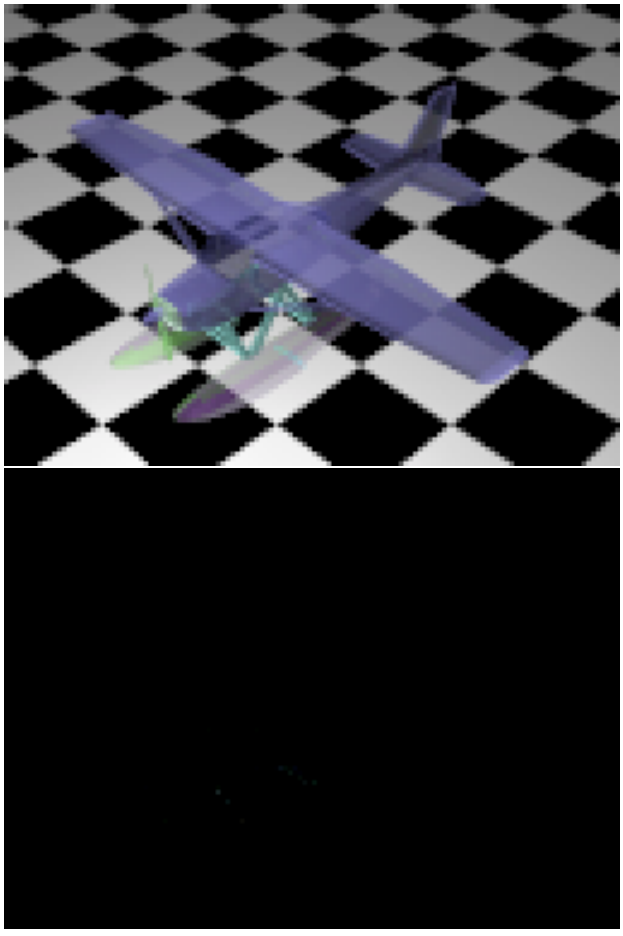

Figure 11: The transparent Cessna seaplane with 16 samples on a 16×16 grid and storage for 4 fragments per pixel. The lower figure gives the per-pixel difference from a 16X sparse supersampled reference image with an unbounded number of fragments.