HELSINKI UNIVERSITY OF TECHNOLOGY
Telecommunications Software and Multimedia Laboratory
29.3.2002
Tik-111.500 Seminar on computer graphics
Spring 2002: Advanced Rendering Techniques

Antialiasing

Seppo Äyräväinen

40431W

Antialiasing

Seppo Äyräväinen
HUT, Telecommunications Software and Multimedia Laboratory

Seppo.Ayravainen@hut.fi

Abstract

This paper describes various aliasing problems in computer graphics, and the techniques for eliminating them, or reducing their visual effect. The theory behind aliasing is briefly introduced. Three main approaches to antialiasing – analytical, A-buffer, and super- and multisampling – are discussed, along with examples of real-world solutions of each. The focus of the paper is on hardware-realizable techniques.

1 INTRODUCTION

Aliasing has always been a problem in raster-based computer graphics. The methods for avoiding its effects have been developed for decades, and many have been successfully used in off-line (i.e. non-real-time) rendering of computer-generated images. During the past ten years, the available processing power has reached the level, where practical antialiasing can also be implemented in real-time applications. This has fueled the research of real-time hardware antialiasing techniques, many of which are discussed in this paper. The focus of the paper is on the so-called full-scene antialiasing methods, so various case-specific techniques, including point and line antialiasing, are left out. Chapter 2 presents the reader with necessary background information, as well as references for getting more information. Chapter 3 discusses antialiasing in the continuous domain, in particular an analytic visible surface algorithm. Chapter 4 introduces the A-buffer (Anti-aliased, area-averaged accumulation buffer) method, as well as its variants and hardware implementations. In chapter 5, supersampling and multisampling – the most popular antialiasing methods today – are discussed. Wherever possible, the use of mathematical formulas is replaced with verbal explanation to make the paper easier to follow. The reader is, however, assumed to be familiar with the fundamental computer graphics terms.

2 BACKGROUND

Most real-world signals are continuous, whereas computers are inherently discrete. In practice this means that the signals need to be sampled, i.e. the signal value has to be saved in discrete points in time or space. This works well in many cases, but there are also signals that cannot be accurately represented with a digital computer. The reasons for this lie in sampling theory, described later in this paper.

Representing a continuous image (or any other signal for that matter) on a computer ideally requires two steps: sampling the image, and reconstructing it from the sampled values for display. For an example, see Foley et al. (1990 p. 620, Fig. 14.9). In practice, both of these steps can introduce problems. Note, that the image to be sampled can be originated from either a source outside the computer, or from a rendering algorithm within a computer.

The simplest method for sampling is known as point sampling. This means simply evaluating the intensity of the image at each pixel's location (at some *X*-*Y* coordinate point within the pixel). However, this method will often fail to capture small details falling between these sampling points (see Foley et al, 1990, p.621). Point sampling can be improved by taking several samples within each pixel, and averaging the values for the final pixel color. This method, called supersampling or multisampling (see chapter 5), causes more of the details to contribute to the pixel. Unfortunately, this approach also has its drawbacks: First, it requires more calculations to be done for each pixel, as well as increases the memory requirements. Second, some signals, especially in computer-generated graphics, have infinite frequencies, that cannot be accurately reproduced, no matter how dense the sampling grid. Examples of infinite frequencies would be a sharp edge in a polygon (abrupt intensity change equals an infinite frequency), or an infinite checkerboard pattern viewed at an angle. No matter how often we sample the signal, there will always be details left between the sample points.

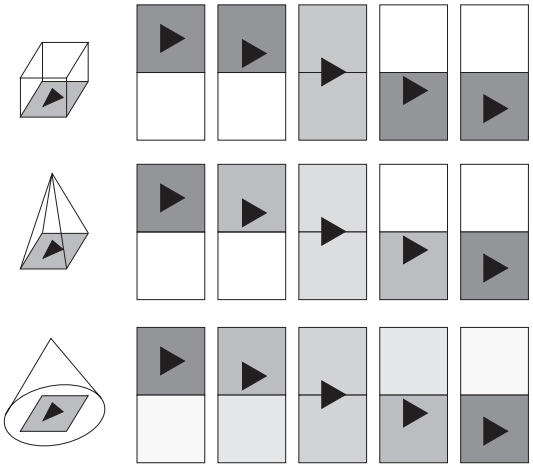

Figure 1: An example of a small black polygon moving across a pixel boundary with white background, using different area sampling methods: unweighted (top), weighted (middle), and weighted with overlap (bottom). Notice the smoother transition of the pixel values in the overlapped case. Adapted from Foley et al. (1990, pp. 622-623).

Another sampling approach, area sampling (Foley et al. 1990 pp. 621-623), avoids the problem of details falling between the sample points. In area sampling, the actual area of each graphical primitive falling inside a pixel is calculated, and these values are averaged for the final pixel color. This is called unweighted area sampling, which means that the whole pixel area is considered as equal. The problem is that if, for example, a small object moves inside a pixel area, the movement does not affect the pixel's color. Only when it crosses the pixel boundary, the color of the pixel (and the neighboring pixel) will change. Weighted area sampling uses a weighting function to emphasize objects near the center of a pixel. This approach makes the color change of

the pixels in the previous example appear more gradual. However, this method still has the drawback, that the pixel's color is only affected by objects falling strictly within the pixel. The result is abrupt changes in pixel colors, when an object enters or exits the pixel area, and is seen as pixel flickering. The solution is to use an overlapping filter, which is centered at a pixel, but has a larger area, and takes into account all the objects falling within it. This way, the abrupt changes in pixel colors are avoided, even when objects cross pixel boundaries. Figure 1 illustrates the sample case of a small black polygon on white background.

2.1 Sampling theory

In addition to observing signals in spatial domain, like as pixels on a screen, they can be observed in the frequency domain. Fourier theory states, that a periodic signal can be seen as a sum of sine waves with different frequencies (the base frequency of a signal and its multiples), amplitudes, and phases. Also non-periodic signals (such as most images), can be seen this way if we consider the whole signal as being one period of a periodic signal. The operations of transforming a signal from the spatial domain to frequency domain and back are called the Fourier transform and the inverse Fourier transform, respectively. Details of these transforms can be found in Foley et al. (1990, pp. 623-628), or in numerous books on mathematics, but they are not essential in understanding antialiasing. More important is to understand that some operations are much easier to do in the frequency domain than in spatial domain, and vice versa.

Sampling theory states, that in order to reconstruct a sampled signal back to its original form, it has to be sampled at more than twice the frequency of the original signal. This minimum sampling frequency is called the Nyquist frequency. In practice, the sampling rate must be even higher because the reconstruction cannot be done with an ideal (infinite) filter. From observing various signals in the frequency domain, it is easily seen that, for example, a pulse function (see Figure 2, top right) has infinite frequency components. For example, sampling a white polygon in a black background creates a similar pulse function for each scanline intersecting the polygon – if the sampling rate is adequate to capture the polygon details. The sampling rate should be infinite to accurately capture the pulse (polygon) boundaries, since otherwise the actual boundary will always fall between two samples.

2.2 Filtering

In order to sample and reconstruct a continuous signal without aliasing, it must either contain no frequencies greater than the Nyquist frequency, or those frequencies must be filtered out before sampling using a low-pass filter. An ideal low-pass filter (a box filter in the frequency domain) cuts out all the frequencies above certain threshold, while not touching the lower frequencies. This is easily seen, since multiplication by a box filter (having the same shape as the pulse function), causes all high frequencies "outside the box" to be clipped out. For an example, see Foley et al. (1990, p. 631, figure 14.21).

In order to apply a low-pass filter, we should first transform the image from spatial to the frequency domain. Fortunately, multiplying the signal with the box filter in the frequency domain equals to convolving it with the inverse Fourier transform of the box

filter in the space domain. Convolution is a fundamental signal processing operation, discussed in many books in the field, and explaining it is out of the scope of this paper. It is best described graphically, with examples found in Foley et al. (1990, pp. 632-633).

More interesting, considering antialiasing, is the selection of the filter for the operation. The inverse Fourier transform of a pulse function, is a *sinc* function, defined as $sin(x)/x$. Unfortunately, this function has nonzero values at infinity (i.e. it has an infinite support), and also has negative values. One could think that truncating the filter would be a good idea, since the values of the sinc function are very small far from the origin, and would not affect the result much. However, this approach causes problems in the form of "ringing" in the image, also known as the Gibbs phenomenon. This causes the Fourier transform of a truncated *sinc* to have ripples at the discontinuities of the pulse function (see Figure 2, bottom). By increasing the *sinc* filter's support, this ringing becomes less noticeable, but its amplitude stays the same. Negative values cause another problem: they may introduce negative values also into the filtered image, which are of course impossible to visualize, and have to be clamped to zero. Practical solutions for the filtering problem are discussed in chapter 5.1.

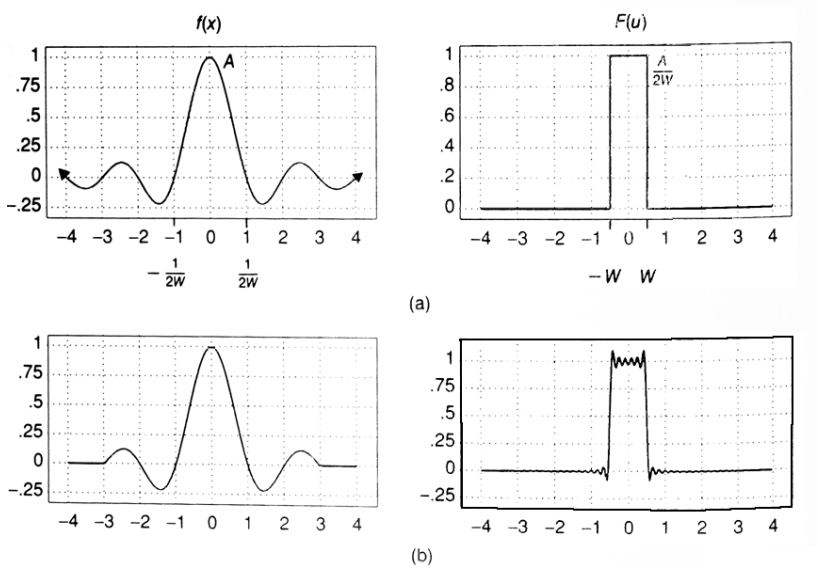

Figure 2: The sinc function (top left), and its Fourier transform (a pulse function, top right). A truncated sinc function (bottom left), and its Fourier transform (bottom right). Notice the ripples in discontinuities of the transform of the truncated sinc. Foley et al. 1990, p.634.

The final stage in the process is *reconstruction*, which means recreating the original signal from its sampled, discrete form. It is discussed in detail in Foley et al. (1990, pp. 636-642). In practice, the reconstruction phase is combined with the low-pass filtering phase, because the final signal is digital and only needs to be defined in discrete points of space (defined by the picture elements on the display device).

2.3 Symptoms of aliasing in computer graphics

Aliasing causes various artifacts in computer-generated graphics, described by Catmull, 1978, including jagged edges (see Figure 4), small objects popping on and off on

successive frames, moiré patterns in periodic images (Figure 3), and fine detail breaking up (Figure 4). The figures below illustrate these artifacts. The nearest neighbor interpolation uses no filtering, and thus is equal to the non-antialiased case, while bicubic filtering closely resembles using a Gaussian filter.

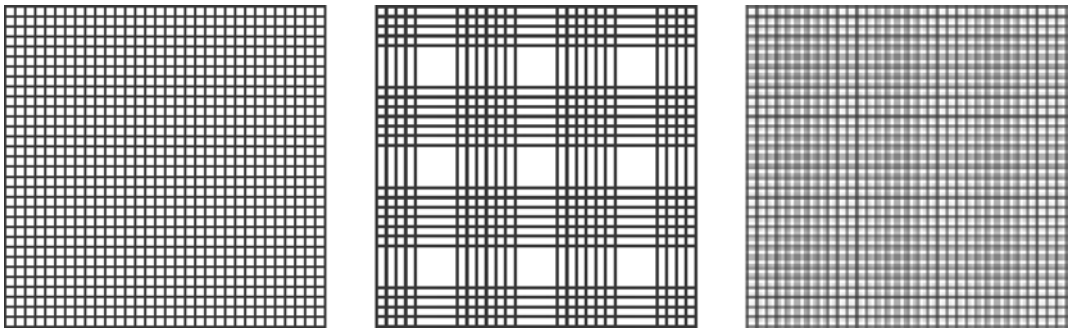

Figure 3: A regular grid (left), resampled from 516×516 to 100×100 pixels in Photoshop, using nearest neighbor (i.e. not filtered, middle) and bicubic (right) filtering. Notice the aliased pattern in the middle image.

Figure 4: A shape (left) resampled from 512×512 to 100×100 pixels in Photoshop, using nearest neighbor (i.e. not filtered, middle) and bicubic (right) filtering. Notice the jagged edges, and detail breaking up in the thinner portion in the middle image.

Aliasing may also occur in time domain, the phenomenon being known as temporal aliasing. Temporal aliasing effects are typically seen in fast rotating objects (such as wheels of a car), which sometimes seem to rotate backwards or slow down, even though that is not the case. The reason for this is basically the same as in other aliasing, i.e. the signal – in this case, motion – is not sampled (i.e. photographed or rendered) often enough. The same artifacts occur in motion pictures and television, but they are reduced by motion blur, caused by the camera shutter. To achieve a similar reduction, artificial motion blur is often used in computer animation.

2.4 Antialiasing methods

Antialiasing methods can be categorized in several ways, none of which are perfect. One of the categorizations is between pre- and postfiltering methods. Since recent research is almost exclusively hardware-oriented, complex and slow prefiltering methods have become more of a curiosity. Another categorization considers analytic methods, edge antialiasing and supersampling, and it is the one used in this paper. Some forget about prefiltering (or analytic) methods completely, and talk about supersampling and A-buffer based methods. In all, a global taxonomy of the antialiasing methods is not yet established.

Prefiltering, described in Foley et al. (1990, p.642), means removing the frequencies of an image, that are above the Nyquist frequency, before sampling. Essentially this means applying a low-pass filter on the image and then sampling. In practice, this involves calculating the area contribution of each polygon for a given pixel. It is the only method that completely removes aliasing. Prefiltering must be done before rasterization (sampling), and therefore cannot as easily be added to a standard renderer like, for example, multi- or supersampling.

Postfiltering methods consider the pixel coverage only at discrete points instead of an exact area calculation. This leads to methods that are easier to implement in hardware, as well as efficient enough to be used in real-time rendering. However, because of their discrete nature, they can never fully eliminate the aliasing. They can only push the aliasing threshold higher in frequency by adding subpixel samples, and possibly trade visible aliasing for noise (by sampling subpixel values pseudo-randomly) or blurring (by averaging subpixel values from several pixels).

The only prefiltering methods discussed in this paper are the Catmull's algorithms (1984, chapter 3 and 1978, chapter 4.1). Postfiltering methods described include supersampling and multisampling. The A-buffer (Carpenter, 1984), and its variations discussed in chapter 4, are sometimes categorized as prefiltering algorithms. However, since their operation is based on discrete coverage masks, they can better be seen as postfiltering methods.

3 ANALYTIC SOLUTION

The analytic visible surface algorithm introduced by Catmull, 1984, solves the visible surface problem for each pixel, while also providing various cinematic effects. It was developed in Lucasfilm™ Ltd within a working group striving to develop a solution for antialiasing, motion blur, and shadows, while handling very high polygon counts (on the order of 80 million). This algorithm was a working solution for all the needs except the shadows, although it is not very efficient by today's standards. Catmull's method is based on filtering the polygons with an analytic Gaussian filter. By changing the shape of the filter (or actually the shape of the polygons under the filter), one can simulate depth of field and motion blur of a real camera.

3.1 The analytic algorithm

The algorithm benefits from labeling objects with an object tag, but it is not absolutely necessary. Also, if possible, contour edges should be marked before processing. In any case, extensive polygon sorting is necessary at the beginning of each frame, making the algorithm quite inefficient.

Polygons are first sorted by their X and Y coordinates using bucket sorting. At each pixel, a list of polygons overlapping the filter is created. The list is sorted using a technique called head sort. This means finding the closest polygon and all the polygons having the same object tag as the closest one. These polygons form the head of the list, which is then sent to a filter routine. At this point, if there are no contour edges in the pixel (filter) area, and the pixel center is covered, the pixel is completely covered and

the filtering begins. If this was not true (i.e. we are at the edge of an object), polygons must be clipped in a phase called the resolver (details found in Catmull 1984, p. 112).

Each pixel has a filter placed at its center. The polygons are convolved with the filter (i.e. their area under the filter is calculated) using an algorithm described by Feibush, 1980. Convolving involves some trigonometry and using a look-up table, pre-calculated based on the shape of the filter.

The filtering can be used for various cinematic effects, such as motion blur and depth of field. Moving a polygon under the filter has the same effect as stretching the filter in the direction of movement. This equals to using a circular filter and shrinking the polygon instead, also in the direction of movement. This creates a motion blur effect. Similarly, an object appearing out of focus can be simulated by making the filter diameter larger, which equals shrinking the polygons under the filter in both X and Y direction. The movement of the polygons is handled by assigning tags, indicating the velocity vectors, to the objects. The algorithm did not correctly handle moving polygons that intersected stationary polygons, but it was not considered an important problem.

4 A-BUFFER ANTIALIASING

A-buffer methods can be considered being a mix of analytic and supersampling methods. One advantage of the A-buffer methods is that they can disregard scanline spans of pixels completely covered by a large polygon, so only the edge pixels need to be considered. This is why they are sometimes referred to as edge antialiasing algorithms. Most of the research of the A-buffer and its descendants has recently been in rendering semi-transparent polygons efficiently. Especially the advanced A-buffer features, such as order-independent rendering of intersecting, transparent surfaces with antialiasing, has been a difficult issue to solve. The details of the hardware architectures are left out of the scope of this paper. Instead, as the A-buffer variants are discussed in historical order, each overview is focused on the improvements made to the previous methods.

4.1 Background: The Catmull's algorithm

Edwin Catmull had introduced the Z-buffer, still used today in practically all graphics hardware, as a part of his Ph.D. thesis, Catmull, 1974 (not freely available). It was a very simple hidden surface method, but had some serious issues: it aliased terribly and didn't handle transparent surfaces. For more information on Z-buffer, see Foley et al. (1990, pp.668-672).

In 1978, Catmull presented an accurate, but computationally very expensive algorithm for both antialiasing and hidden surface calculation, in the continuous domain. In the algorithm, visible polygon fragments falling inside a given pixel are first determined by extensive sorting (details found in Catmull 1978, pp.7-9), and then clipped to pixel boundaries. The visible subpixel area of each fragment is then calculated, and multiplied by its color value. Finally, these are summed to form the final averaged color of the pixel. Since the algorithm only needed to consider polygon edges, it suited well Catmull's purposes of rendering 2D cartoons with large, flat-shaded polygons.

4.2 A-buffer

Loren Carpenter wanted to improve Catmull's methods, as he stated in his paper (1984, p.104): *"What is needed is a method that combines the simplicity of the Z-buffer with the two-dimensional antialiasing benefits of Catmull's full polygon process at each pixel."*

The A-buffer, introduced by Carpenter in 1984, approximated Catmull's continuous-domain algorithm by using discrete subpixel samples instead of exact area calculation. This approach was much easier to efficiently implement in hardware, since subpixel fragment areas, defined by 4 by 8 bit masks, could be calculated using simple XOR (exclusive or) operations instead of the floating required in analytic methods. The method for creating the final mask for a fragment is illustrated in Figure 5.

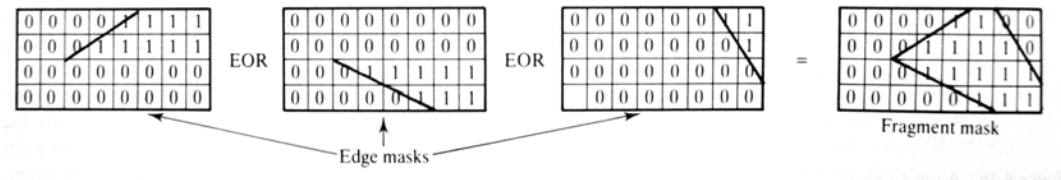

Figure 5: Construction of A-buffer fragment mask from masks of its edges. Exclusive-or is denoted EOR in the figure. Watt and Watt (1992, p.128).

A-buffer keeps in memory linked lists of fragments within each pixel. Fragments are sorted by the frontmost *Z* value to support transparency. Fragments should also be assigned an object tag for efficient processing, since they can be merged to save memory if they overlap in *Z* and belong to the same object. In addition to the minimum *Z* value, a maximum *Z* is also kept for each fragment. This is needed to approximate rendering of intersecting surfaces. However, since this approximation assumes that fragments overlapping in *Z* have opposite-signed slopes, it fails in some cases (see chapter 4.5 for examples). After the fragments are processed, their contribution to the pixel's final color is calculated, starting from the frontmost fragment, and using the subpixel mask values.

A disadvantage of the A-buffer, considering its implementation in hardware, is that there is no limit in how large the lists of potentially visible polygons can grow. Also, the work needed per pixel is not known in advance. Although there are some hardware implementations of the A-buffer, it is mostly a software solution for high-quality rendering. It was originally developed for a scanline rendering system called REYES™ (Renders Everything You Ever Saw) at the Computer Division of Lucasfilm Ltd. This company later became known as Pixar™, and REYES is the underlying rendering engine of their implementation of Renderman Interface used to produce many full-length computer animations.

4.3 EXACT

The EXACT (Exact Area Coverage calculaTion) algorithm by Schilling and Straßer, 1993, solves the A-buffer's problem with intersecting surfaces. Schilling and Straßer (1993, Figure 6, p. 87) introduce priority masks, which keep track of which of the

intersecting surfaces is the frontmost in which part of the subpixel mask. Priority masks are used to modify the pixel's coverage mask, if needed. This approach handles antialiasing in most cases of intersecting surfaces.

Priority masks can be included to enhance a software A-buffer renderer, and the paper presents a hardware implementation for calculating the masks efficiently. The introduced hardware also performs the list processing needed in A-buffer algorithms.

4.4 Modified A-Buffer

The modified A-buffer algorithm of Apple Computer™, described by Winner et al. 1997, offers antialiasing and order-independent transparency, but may require multiple passes to handle the primitives. If there are *n* primitives per pixel, and they cannot be combined, *n* passes are required. Modified A-buffer's key advantage is that it does not require extra memory and typically only degrades performance 40% compared to non-antialiased case. However, the need for multiple geometry passes can be considered a bad thing for performance.

A 4×4 staggered subpixel mask with 8 sample positions is used for the fragment coverage calculations. The screen is partitioned in 16×32-pixel blocks, which are processed one at the time. This memory optimization allows both the Z- and the frame buffers to be stored on-chip for efficient calculation, while only degrading performance to a reasonable extent when antialiasing is not used.

4.5 Z^3

Z^3 by Jouppi and Chang, 1999, is a hardware A-buffer technique that takes quite a different approach from Apple's system. Instead of trying to save memory, it uses a fixed amount of extra memory per pixel to accomplish both antialiasing and order-independent transparency. Unlike the basic A-buffer, it correctly handles interpenetrating transparent surfaces.

Z^3 only keeps track of a fixed amount of subpixel fragments for each pixel. This makes hardware design easier, since the memory requirements are known in advance. In addition to a normal *Z* value, *Z* slopes (gradients) in both *X* and *Y* directions are saved for each fragment. The name Z^3 stems from the fact that three-dimensional *Z* information is used for each fragment.

Keeping track of the Z slopes is important, since there are many cases when incomplete *Z* information can cause errors in visibility and antialiasing calculations (Jouppi and Chang 1999, p. 86):

1. Single *Z* at pixel center – Fails in cases like the top left in Figure 6. Fragment B would be visible, even though the opposite is true.

2. *Z*min and *Z*max – Used in the original A-buffer algorithm. Assumes, that the slopes of the fragments are opposite, so cases like in top right of Figure 6 fail. In this case, A and B would be blended, although only A should be visible.

3. Fragment subpixel *Z* average – Calculating the average of the sample point *Z* values for each fragment handles cases like the top left, but still fails in a case like the bottom left in Figure 6. If B is moving towards the viewer, the whole pixel will suddenly change its color to B's color.

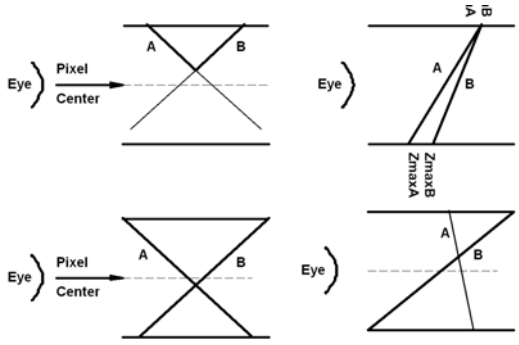

Figure 6: Difficult cases for non-complete Z information. Jouppi et al. (1999, p.86).

4.6 R-Buffer

The R-buffer (Recirculating fragment buffer) was introduced by C.M. Wittenbrink in 2001. It is primarily an efficient hardware solution for order-independent transparency, but also supports A-buffer antialiasing. The R-buffer is a first-in-first-out (*FIFO*) buffer, storing fragments not fitting into the Z-buffer for a given pixel. Since the R-buffer is circulating, there is no need to waste time in moving data within the buffer, but only to change pointer values. If the buffer overflows, excessive fragments are paged to main memory.

Each pixel has a state machine, needed for compositing opaque and transparent fragments. In addition to a unique R-buffer, the method utilizes a second Z-buffer and three bits per pixel for storing state information. The ordinary Z-buffer is used to store either the closest opaque sub-pixel fragment, or the furthest transparent one. Other fragments are either put into the R-buffer or discarded, depending on the current pixel state.

To support order-independent transparency, the R-buffer requires multiple rendering passes, like the Modified A-buffer method in chapter 4.4, but only one geometry pass. If there is enough memory, it can be traded for the passes required. If n is the worst-case transparent layer depth complexity, the passes required varies from $2n$ (in the case of no extra frame buffers) to n / N (in the case of N frame buffers). In the three test cases described in Wittenbrink (2001, p.78), the R-buffer method required from 2.13 to 3.63 times more memory than a normal Z-buffer.

5 SUPERSAMPLING AND MULTISAMPLING

Supersampling and multisampling are the simplest antialiasing methods, and also the easiest to implement in hardware. The distinction between these methods is somewhat unclear, and some implementations can be seen as belonging to either of the categories. Both methods, however, belong to the postfiltering category, since filtering is done after rendering.

In supersampling, the image is first rendered in a larger resolution, and then scaled down to the final size. In multisampling, each pixel is sampled from several locations, and those samples are used to reconstruct the final value for each pixel. Both of these methods essentially require taking many samples for each pixel in the final image. The image quality achieved depends both on the number and the pattern of the samples taken per pixel (see chapter 5.2).

Unfortunately, as mentioned before, neither supersampling nor multisampling can completely remove aliasing from an image. No matter how dense the sub-pixel sampling is, there can always be frequencies higher than the samples can capture. Essentially, the Nyquist (aliasing) limit is only pushed higher by increasing the sampling frequency. If the signal is not low-pass filtered before rendering, most computer-generated images are likely to alias. However, especially in real-time rendering applications, both methods produce acceptable images considering the low performance cost, and are therefore used in almost every hardware antialiasing implementation.

5.1 Supersampling process in general

In supersampling, the process is generally as follows:

1. Create a virtual image at a higher resolution than the final image.

2. Apply a low-pass filter.

3. Resample the filtered image.

Step 1 is easy, but requires a lot of memory and bandwidth. Step 2 is discussed earlier, in chapter 2.2. To further optimize the process, steps 2 and 3 can be combined by only creating the filtered image at the new sample points, since there is no need to actually recreate an analog signal from the samples.

As discussed earlier, the *sinc* function cannot be used in practice. A box filter, which equals a simple average of the sample points, also known as unweighted area sampling, would be easier to use. Unfortunately it is very bad as a filter, since it passes infinitely high frequencies and attenuates the desired frequencies. In practice, $sinc^2$ and *Gaussian* filters are much better. $Sinc^2$ is the Fourier transform of a *triangle function,* and the Gaussian function is a Fourier transform of itself. Examples of these are found in Foley et al. (1990, p.635, figure 14.25 (b) and (c), respectively). Both attenuate fast, and are thus quicker to compute. Using the $sinc^2$ filter is better known as bilinear filtering.

5.2 The choice of sampling pattern

Increasing the number of sample points will improve the image quality, but also increase the memory requirements and decrease performance. Therefore, careful selection of sampling points is a very significant factor in practical super- and multisampling. Regular sampling patterns are not good in removing the aliasing effects most noticeable to human eye, since humans are very sensitive to regular patterns.

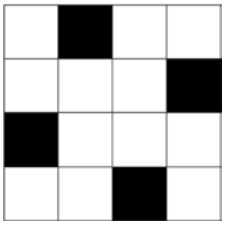

Figure 7: An example of a 4×4 grid with 4 sample points (black squares). This pattern gives 5 intensity values in vertical, horizontal, and diagonal directions, while using only 4 samples per pixel.

If a regular grid is used, best results considering memory use and performance are achieved by taking samples only from part of the possible sample locations, for example, at n locations when using an $n \times n$ grid (see Figure 7). This gives $n+1$ different intensities for edges in most directions. This technique, called sparse sampling, also allows using a finer sampling grid while using less memory and bandwidth.

5.3 Stochastic sampling

A technique called stochastic sampling, described in detail in Cook, 1986 and Dippé and Wold, 1985, utilizes the fact that the human visual system is more sensitive to regular than noisy aliasing patterns. This method is mostly used in ray-tracing, where casting a ray can be easily done anywhere within a pixel's area. Sampling is done using a random-like pattern, some of which are shown in Figure 8. The Poisson pattern is generated simply by adding points at random locations. Poisson disc is much more expensive to calculate, since there is a limit on how close a point can be relative to others. Generating a Poisson disc pattern thus requires saving the locations of each point. If a new point is added, and fails the minimum distance test, it must be discarded, and a new location generated. The jitter pattern is generated by using a regular grid, and then jittering (i.e. moving) each point by a random amount in X and Y direction.

The Fourier transforms of these patterns indicate the distribution of noise in the frequency domain of the final image. Poisson pattern generates roughly equal amount of both high and low frequency noise, which is not good because low frequency noise is more easily noticed, and thus not desirable. Poisson disc avoids noise in low frequencies very well, but since it is expensive to calculate, jittering seems to offer the best "price-performance" ratio.

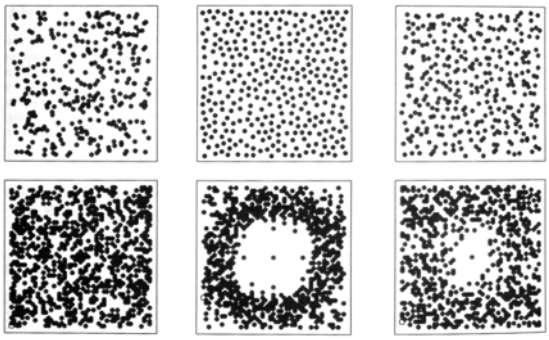

Figure 8: Noise patterns (upper), and their Fourier transforms (lower). From left to right: Poisson, Poisson disc, and Jitter. Watt and Watt, 1992, p. 134.

5.4 Hardware multi- and supersampling implementations

ATI™ SmoothVision™, described in ATI, 2001, uses groups of 16 sample points, which are divided among varying number of pixels, depending on the desired antialiasing quality. In 2× mode, the samples cover 8 pixels, in 4× mode 4 pixels, and so on. Each pixel has 8 alternative pre-programmed, jittered sampling locations, which gives the sampling pattern a pseudo-random look (see Figure 9). The apparent randomness of the sample points creates images that are more pleasing for the eye, since aliasing is traded in for noise. ATI claims that their technique is a supersampling one, and that it produces better-looking images than the multisampling approach of NVIDIA™, since they look up texture information at each sample point, instead of only once per pixel.

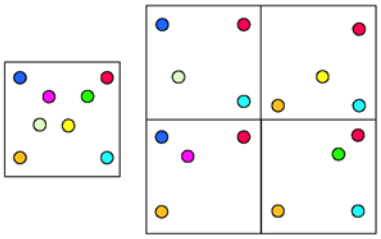

Figure 9: The eight possible sample locations for a single pixel (left), and possible pixel sample locations for the SmoothVision 4× setting (ATI, 2001).

The SGI™ RealityEngine™, described by Akeley, 1993, uses full 4×4 supersampling with a regular grid, which is bad for both performance and visual quality. There are two options for sampling: point sampling and area sampling. In point sampling, only the samples strictly inside a fragment are considered. In area sampling, a method related to the one described by Schilling, 1991, is used. In practice this means that the actual area of a fragment is related to the number of sample points used, even if the samples are not inside a fragment (see Figure 10). According to Winner et al. (1997, chapter 2.5) the RealityEngine uses *X*- and *Y*-slope values, and a single depth sample per layer to reconstruct sub-pixel depth values.

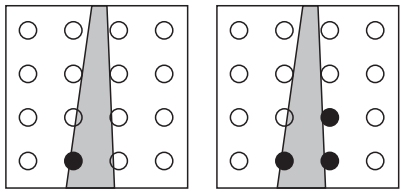

Figure 10: Sampling a fragment using point sampling (left) and area sampling (right). Circles represent the sampling positions, of which the black ones are chosen. Adapted from Akeley (1993, p.113).

The SGI InfiniteReality™, introduced by Montrym et al. 1997, was built from the beginning for one purpose only: to realize the whole OpenGL™ pipeline in hardware. It consists of a huge amount of application specific integrated circuits (ASIC), each dedicated for a single stage of the pipeline, and each phase massively parallelized. It uses 8×8 sparse supersampling for antialiasing – much better than the RealityEngine in that sense. Each primitive is sampled from 1, 4, or 8 of the 64 possible locations. The *Z* information of all the 64 subsamples is kept for accurate hidden-surface calculation. The current implementation is called InfiniteReality3, but there are no major architectural differences to the original IR. More information can be found on the InfiniteReality technical report, SGI, 1997 (a HTML translation is available at http://www.futuretech.vuurwerk.nl/ir_techreport.html).

Figure 11: NVIDIA Quincunx™ sample points (left) and shifted points of Accuview™ (right).

In the 2× multisampling mode of the NVIDIA GeForce™ 3 and 4 series, the image is drawn into two separate locations in the frame buffer. Between the two drawing passes, the sampling point is moved half a pixel. These samples are then filtered to form the final pixel. In normal mode, only one sample of each pass is considered per pixel. In the Quincunx mode, a sample from one pass is filtered along with 4 surrounding samples from the other pass (see Figure 11, left). In Accuview enhancement, found in GeForce 4 only, the sample positions are slightly shifted for more accurate sampling (see Figure 11, right). There is also a 4× mode available. Since these methods causes blurring of textures, in GeForce 4 they are combined with anisotropic texture filtering, the description of which is out of scope of this paper. Effectively, anisotropic filtering reduces blurring of textures on surfaces that are not facing the viewer. More information can be found in the NVIDIA Accuview technical brief, 2002 (http://www.nvidia.com/docs/lo/1451/SUPP/accuview.final.pdf).

In the antialiasing method used by 3Dfx™ Voodoo™ 5, the image is first rendered into 2 or 4 separate back buffers, jittering the image at sub-pixel level for each buffer. The buffers are swapped, after which each pixel is averaged from the buffers using a custom multi-input antialiasing RAMDAC (Random Access Memory Digital-to-Analog Converter). This method is interesting, because the final image does not exist at all in digital form, but is generated on the fly into an analog signal in the video circuitry. It is referred to as both multisampling and supersampling approach, depending on the

source. The Voodoo series was discontinued shortly after NVIDIA bought 3Dfx's patents in December, 2000.

The Wildcat™ series from 3Dlabs™ uses a supersampling optimization called Superscene Antialiasing. It uses a 16×16 grid, from which 2, 4, 8, or 16 samples are taken. The advantage of the system is to be able to dynamically adjust the number of sampling points, depending on how many fragments there are in the current pixel. This approach saves both memory and bandwidth. More information can be found from http://www.3dlabs.com/product/technology/superscene_antialiasing.htm.

6 CONCLUSIONS

Implementing both accurate and efficient antialiasing is still a challenging problem in today's computer graphics. However, recent generations of consumer graphics cards are already capable of doing decent antialiasing in real-time. The line between off-line and real-time graphics is fading away quite rapidly, as the computational power and available texture and frame buffer memory is roughly doubled at consumer level every year. Memory bandwidth currently seems to be the biggest limitation in the real-time graphics performance. However, it seems that multisampling is the technique that will rule the market for at least some years, since the solutions from many leading manufacturers, including NVIDIA and SGI, are currently using it. A-buffer variants, especially the Z^3 and the R-buffer (developed by Compaq™ and Hewlett-Packard, respectively), show great potential, and create high-quality images in hardware. Those methods will likely be primarily used in professional workstations, where image accuracy is crucial. Consumer applications are currently more focused on performance, and it will probably take some time before we see any A-buffer based cards in consumer price range.

REFERENCES

ATI SmoothVision white paper, ATI technologies, Inc. 2001. *http://www.ati.com/na/pages/technology/hardware/smoothvision/smoothvision.pdf.*

Carpenter, L.C. 1984. The A-buffer, an Anti-aliased Hidden Surface Method. *Computer Graphics (Proceedings of the ACM SIGGRAPH '84 Conference)*, Vol.18, No.3, pp.103 – 108.

Catmull, E. 1974. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Ph.D. thesis, Department of Computer Science, University of Utah, December 1974.

Catmull, E. 1984. An Analytic Visible Surface Algorithm for Independent Pixel Processing. *Computer Graphics (Proceedings of the ACM SIGGRAPH '84 Conference)*, Vol.18, No.3, pp.109 – 115.

Cook, R. 1986. Stochastic sampling in computer graphics. *ACM Transactions on Graphics*. Vol. 5, No. 1, pp.51 – 72.

Dippé, M.A.Z. and Wold, E.H. 1985. Antialiasing through stochastic sampling. *Computer Graphics (Proceedings of the ACM SIGGRAPH '85 Conference)*. Volume 19, No.3, pp. 69 – 78.

Feibush, E; Levoy, M., Cook, R. 1980. Synthetic Texturing Using Digital Filtering. *Computer Graphics (Proceedings of the ACM SIGGRAPH '80 Conference)*, Vol.14, No.3, pp.294 – 301.

Foley, J. D.; Van Dam, A.; Feiner, S. K.; Hughes, J. F. 1990. *Computer Graphics: Principles and Practice.* Second edition. Menlo Park, CA. Addison-Wesley. 1175 p.

Jouppi, N.P. and Chang, C. 1999. Z^3: An Economical Hardware Technique for High-Quality Antialiasing and Transparency. *Proceedings of the 1999 Eurographics / SIGGRAPH Workshop on Graphics Hardware,* pp.85 – 93.

Molnar, S.; Eyles, J.; Poulton, J. 1992. PixelFlow: high-speed rendering using image composition. *Computer Graphics, (Proceedings of SIGGRAPH '92),* Vol. 26, No. 2, pp. 231 – 240.

Montrym, J.S.; Baum, D.R.; Dignam, D.L.; Midgal, C.J. 1997. InfiniteReality: A Real-Time Graphics System. *In Proceedings of the 24th annual conference on Computer graphics and interactive techniques, 1997 (SIGGRAPH '97),* pp. 293 – 302.

Schilling, A. 1991. A New Simple and Efficient Antialiasing with Subpixel Masks. *Computer graphics (Proceedings of SIGGRAPH '91),* Vol. 25, No. 4, pp.133 – 141.

Schilling, A. and Straßer, W. 1993. EXACT: Algorithm and Hardware Architecture for an Improved A-Buffer. *In Proceedings of the 20th annual conference on Computer graphics and interactive techniques, 1993 (SIGGRAPH '93),* pp.85 – 91.

Watt, A. and Watt, M. 1992. *Advanced Animation and Rendering Techniques: Theory and Practice.* New York, NY. Addison-Wesley. 455 p.

Winner, S.; Kelley, M.; Pease, B.; Rivard, B.; Yen, A. 1997. Hardware accelerated rendering of antialiasing using a modified a-buffer algorithm. *In Proceedings of the 24th annual conference on Computer graphics and interactive techniques, 1997 (SIGGRAPH '97)*, pp.307 – 316.

Wittenbrink, C.M. 2001. R-buffer: A pointerless A-buffer hardware architecture. *In Proceedings of the 28th annual conference on Computer graphics and interactive techniques, 2001 (SIGGRAPH '01)*, pp.73 – 80.