An Efficient Low-Cost Antialiasing Method Based on Adaptive Postfiltering

Rynson W. H. Lau

***Abstract*—Aliasing in computer-synthesized images not only limits the realism of the images, but also affects the user's concentration. Many antialiasing methods have been proposed to solve this problem, but almost all of them are computation intensive, and some of them are also memory intensive. While these may not be limitations for high-end applications such as medical visualization and architectural design, this kind of antialiasing methods may still be far too costly for the low-cost applications. In this paper, we propose an antialiasing method that operates in the image domain. It is based on fitting curves to the discontinuity edges extracted from the aliased images to reshade those edge pixels. (Note that a curve may be considered as a general form of lines.) To improve the performance and the simplicity of the method, we propose to preprocess all possible edge patterns and fit curves in advance. During runtime, we only need to construct an index to obtain the filtering information from a lookup table. The new method is extremely simple and efficient. It provides a very good compromise between hardware cost and output image quality. In addition, because the new method has a very low computational cost, and hence low power consumption for hardware implementation, it is particularly suitable for low-cost mobile applications such as computer game consoles and palm computers, where low implementation cost and low power consumption are important design factors.**

***Index Terms*—Adaptive smoothing, antialiasing, image enhancement, image postfiltering.**

I. Introduction

VISUAL artifacts, or *aliasing*, which usually appear in computer-synthesized images, not only limit the realism of the output images, but also affect the viewer's concentration. These artifacts are mainly caused by the point sampling approach adopted by most rendering engines, where if a polygon covers the sample point of a pixel, it is assumed to cover the whole pixel and if it does not cover the sample point, it is assumed not covering the pixel at all. This results in visual artifacts that often appear in the form of jagged edges, strange patterns or abnormal flashing in the output video images. In some applications, in particular real-time applications where a user must give instant response, this aliasing problem may actually cause the user to make wrong judgements/decisions. Many methods have been proposed to solve the aliasing problem, but most of them suffer from high computational and memory costs. Although such limitations may not really be a problem for high-end applications, like medical visualization and architectural design, these methods may be too costly for low-cost systems, such as computer game consoles and palm computers. This explains why most, if not all, computer game consoles do not consider the aliasing problem at all.

In this project, our objective is to develop a low-cost antialiasing method that would reduce the aliasing problem of output video images in the mobile game console and palm environments with as little increase in computational and memory costs as possible. This paper presents such a method based on adaptive postfiltering. The main idea of this method comes from the image-processing domain instead of the traditional graphical domain. The method is extremely robust and efficient. It is also very simple and cheap to implement. In mobile game consoles and palm computers, power savings is a very important design issue. Although the price for PC graphics accelerators has come down quite dramatically in recent years, these accelerators still suffer from high power consumption due to the complexity of the circuitry in order to provide the rendering performance. Traditional antialiasing techniques such as supersampling are still far too expensive to be used there. As will be shown in Section VII, our method has very low computational cost, even when written in high-level language. Application of it in the mobile game console environment would immediately improve the visual quality with very little increase in computational cost, and therefore battery power consumption.

The rest of the paper is organized as follows. Section II surveys existing work on antialiasing. Section III presents the main idea of our method. Sections IV and V explain how we detect discontinuity edges and construct the curve-fitting table, respectively. Section VI presents the block-level architecture of the method. Section VII demonstrates and discusses some experimental results of the method. Section VIII critically compares the proposed method with some popular antialiasing methods. Finally, Section IX briefly concludes the paper.

II. Relevant Work

Most image-generation methods suffer from the aliasing problem. For example, the z-buffer method [5] is widely implemented in most of the existing hardware graphics accelerators for hidden surface removal. However, because of its point sampling nature and its low sampling resolution (typically once per pixel), the output images usually contain some aliasing artifacts. Many methods have been proposed to solve/reduce this aliasing problem. They can be roughly classified into two approaches based on our context here: the traditional approach and the postfiltering approach. The traditional approach attempts to solve the aliasing problem as the image is being

Manuscript received September 7, 2001; revised November 12, 2002. This paper was recommended by Associate Editor A. Kot.

The author is with the Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong, and also with the Department of CEIT, City University of Hong Kong, Hong Kong (e-mail: rynson@cs.cityu.edu.hk).

Digital Object Identifier 10.1109/TCSVT.2003.809825

rendered, while the postfiltering approach does it after the image is rendered.

A. Traditional Approach

Most existing hardware graphics accelerators are implemented based on the z-buffer method [5]. However, the z-buffer method is known to suffer from aliasing, due to its point sampling nature and its low sampling rate. To solve the aliasing problem, most systems apply the supersampling technique [1], [2], [7], [14] by increasing the number of sample points per pixel and then filtering all the sampled values within each pixel down into a single pixel value. Although the supersampling method is a simple extension of the original z-buffer method [5], it requires much computation time to process and a large memory space to store the extra samples.

To reduce memory cost, the accumulation buffer method [10] renders the same scene repeatedly at different subpixel positions and accumulates sampled values in an accumulation buffer. The limitation of this method, however, is the increase in processing time, which is proportional to the subpixel resolution. On the other hand, the SGI RealityEngine [1] and InfiniteReality [15] use the sparse mask method to lower the number of samples per pixel. Instead of sampling all the subpixel locations within a regular subpixel grid of a pixel, they allow the application to select any 4, 8, or 16 sample locations from an 8×8 subpixel grid for antialiasing. However, these systems still require a considerable amount of memory and expensive hardware.

As it is noted that aliasing occurs mainly at pixels with polygon edges, some methods have been proposed to focus the rendering effort only on polygon edges in order to reduce the processing time. In [6], [11], and [21], the color contribution of each polygon at a pixel is estimated by the area of the pixel covered by the polygon edge. This requires accumulation of polygon strips in a temporary buffer. We have recently developed an adaptive supersampling method for distributed rendering [13]. The method supersamples only pixels with polygon edges. The sampling resolution may change from frame to frame according to the memory usage statistics of previous frames. Although the memory usage of these methods is generally lower than that of the supersampling methods, they require dynamic memory allocation, and the processing cost may still be too high for low-cost systems.

Another type of method to reduce both processing time and memory usage is to use a subpixel mask to approximate the polygon visibility within a pixel. The A-buffer method [4] accumulates all visible polygon edges in a pixel in the form of a depth-sorted fragment list. Each fragment contains a subpixel mask, indicating the pixel coverage of one visible polygon. To reduce memory usage and processing time, only one color value and two depth values, representing the minimum and maximum depths, are stored in each fragment. However, the method may not be able to compute the visibility correctly when two fragments overlap in depth. In addition, the paper [4] does not address how to compute the subpixel masks with correct coverage. To address the first problem, [19] proposes a method to compute the priority mask, which indicates which part of each polygon is in front, when two polygon edges overlap in depth. To address the second problem, [18] proposes an efficient method to compute the subpixel masks with correct coverage, in particular, when the polygon edge is nearly horizontal or vertical. In [20], a method for computing the subpixel masks is proposed to further improve the quality of the output image through subpixel relocation. Instead of storing the subpixel masks, we earlier proposed a method to store only the positions where a polygon edge crosses a pixel and a method to index the appropriate subpixel mask from a precomputed table given the positions [12]. Although all these methods can significantly reduce memory usage, they suffer from two limitations. First, they require dynamic memory allocation and the number of fragments in each node is unpredictable. Second, they require fragment sorting at each pixel according to the depth value. As a result, hardware implementation of these methods is very complex and expensive.

B. Post-Filtering Approach

A different kind of antialiasing methods is based on postfiltering. After an image is rendered, problem pixels are reshaded to reduce the aliasing effect. Such methods provide in general a cheaper way of antialiasing images because only visible objects need to be processed. One simple method of this approach is to apply a Gaussian filter to those pixels with high local contrast. However, such a method will simply blur the object edges without considering the geometric shapes and orientations of the edges being smoothed.

Bloomenthal proposes a more interesting method [3]. First, high-contrast pixels in the image are detected and linked. Straight lines are then fitted to approximate the geometric shapes of the edges. Smoothing is performed by considering how much a fitted line covers each edge pixel. This method produces a considerably better result but it suffers from two limitations. First, smoothed objects may have a polygonal look because line fitting has only zero-order continuity. The second limitation is more serious. Although the idea of the method is very simple, the algorithm is extremely complex. This is because edge patterns can occur in many different forms. Tracing the edge pixels and determining appropriate lines for fitting at runtime are extremely difficult to do.

Another method is proposed called discontinuity edge overdraw [17]. In this method, discontinuity edges, which include silhouette edges and sharp edges (edges that separate polygons of different colors), are first identified. After the image is rendered, the discontinuity edges are then redrawn as antialiased lines. Since most hardware graphics accelerators available in the market support the drawing of antialiased lines, this rendering of the antialiased lines is extremely efficient. However, this method has two limitations. First, the overdrawn lines will increase the size of the objects by a small amount (typically half of a pixel). This affects the accuracy of the redrawn objects. Second, the preprocessing overheads in determining the silhouette edges and sharp edges of each model are very high, limiting the application of the method to scenes with rigid objects only.

Our method solves the two problems of Bloomenthal's method in two ways. First, instead of using line fitting, which has the continuity problem between the fitted lines, we fit curves or lines to the edge pixels, depending on the local edge curvatures, to produce a smoother contour. Second, instead of fitting curves to the edge pixels at runtime, we use the local

edge pattern to construct an index to look up the information for smoothing from a precomputed table. A further advantage of our method as compared with Bloomenthal's method is that our method is a local filtering method, and hence, it is suitable for parallelization. As will be shown in Section VII, our method is extremely efficient and does not require preprocessing, in contrast with [17].

III. Antialiasing With Curve Fitting

When the contrast between the foreground object and the background is small, we may not notice aliasing. However, when the contrast is high, aliasing may become obvious. To solve the aliasing problem in a postfiltering manner, after the image is rendered, we detect pixels with high local contrast (referred as *discontinuity edge pixels* or simply *edge pixels*). These pixels are our target for postfiltering to reduce the aliasing effect.

To extract edge pixels from an image $I(x, y)$, we apply a second-order differentiator $E(x, y)$, followed by an edge detector $F(x, y)$, to $I(x, y)$. Another function $H(x, y)$ is applied in concurrent with $F(x, y)$ to determine the edge pixels that need to be reshaded

$$E(x, y) = \frac{\delta I^2(x, y)}{\delta x^2} + \frac{\delta I^2(x, y)}{\delta y^2}$$

$$F(x, y) = \begin{cases} 1, & \text{if } ZCrossing(x, y) \\ 0, & \text{otherwise} \end{cases}$$

$$H(x, y) = F(x, y) \quad \text{AND} \quad HContrast(x, y)$$

where $ZCrossing()$ returns 1 (or TRUE) if $E(x, y)$ indicates a zero-crossing pixel and $HContrast()$ returns 1 (or TRUE) if $I(x, y)$ is a high-contrast pixel.

Since a discontinuity edge in an image divides the local pixels into two regions, we refer to one region as the foreground region and the other as the background region, for the sake of clarity. (Of course, these two regions may, in fact, represent two neighboring color regions of the same object.) Hence, to reshade a given edge pixel $P(x, y)$, we may trace the local edge contour connected to $P(x, y)$ and fit a curve to the contour to approximate the percentage of $P(x, y)$ covered by the foreground region, α_f. The percentage of $P(x, y)$ covered by the background region α_b can then be determined as $\alpha_b = 1 - \alpha_f$. This α_f may then be used to reshade $P(x, y)$. However, tracing the edge pixels and fitting curves to them are very expensive processes to execute in runtime. In particular, there are many exceptional cases that need to be considered when tracing the edge pixels. To improve the performance of the antialiasing method, our idea is to consider all possible edge patterns and determine the appropriate curves for fitting in advance in order to minimize the amount of computations in runtime.

Once the high-contrast discontinuity edge map $H(x, y)$ is constructed, we may reshade each edge pixel according to the local edge pattern of the map. Hence, the output image $I'(x, y)$ can be defined as

$$I'(x, y) = \begin{cases} S(x, y), & \text{if } H(x, y) \text{ equals 1} \\ I(x, y), & \text{if } H(x, y) \text{ equals 0} \end{cases}$$

where $S(x, y)$ is a reshading function that computes a new color for pixel $P(x, y)$ as follows:

$$S(x, y) = \alpha_f[T(x, y)] * I_f(x, y) + (1 - \alpha_f[T(x, y)]) * I_b(x, y)$$

where $I_f(x, y)$ and $I_b(x, y)$ represent the foreground and background colors, respectively, for reshading pixel $P(x, y)$. The locations to obtain the colors can be indexed from the precomputed table as follows:

$$I_f(x, y) = I(x + x_f[T(x, y)], y + y_f[T(x, y)])$$

$$I_b(x, y) = I(x + x_b[T(x, y)], y + y_b[T(x, y)])$$

where $x_f[]$ and $y_f[]$ are the precomputed relative x and y locations, respectively, from $P(x, y)$ to obtain the foreground color value. $x_b[]$ and $y_b[]$ are similarly defined to obtain the background color value. $T(x, y)$ is the index to the pre-computed curve table. It is defined as follows:

$$T(x, y) = \bigotimes_{k=y-n}^{y+n} \bigotimes_{j=x-m}^{x+m} F(j, k)$$

where $\bigotimes$ represents an ordered concatenation function. $(2m + 1) \times (2n + 1)$ defines the size of the local region.

We describe in detail how this idea can be implemented in the following sections.

IV. Detecting Discontinuity Edges

There are many ways to implement the discontinuity edge detector. Since most real-time rendering engines use Gouraud shading to compute pixel colors, the output images are often very smooth and the simple Laplacian filter [16] is good enough for our application. The filter mask that we use is shown as follows:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}.$$

However, since we notice aliasing only when the local contrast is high, we add a thresholding step after the Laplacian convolution. All zero-crossing pixels with a local contrast higher than a user defined threshold value will be marked as edge pixels for reshading. In order to prevent fragmentation of edges due to this thresholding process, all zero-crossing pixels with a local contrast lower than the threshold value will also be marked as edge pixels but they will not be reshaded, and hence their intensity values will not be modified. These pixels are only used to improve the accuracy of the curve-fitting process in reshading the high-contrast edge pixels.

Usually, a zero-crossing occurs between two pixels. We need to decide which of the two pixels should be marked as an edge pixel. For a natural image captured with a camera, because the image is already low-pass filtered, we may simply mark the pixel with a filtered value closer to zero as the edge pixel. However, this method does not work here because the rendered images are not low-pass filtered. We have considered two approaches here. The first approach is to work on *dual lattice* by assuming that the discontinuity edge crosses around the center of the pixel pair. To implement this scheme, we may just

mark one of the two pixels. For a horizontal pair, we always mark the one on the left, and for a vertical pair, we always mark the one at the top. After we have computed the fitted curve, we may shift this curve half of a pixel to the right for horizontal pair or to the bottom for vertical pair. This approach, in general, produces more accurate edges after reshading. However, in most cases, it requires reshading both pixels of the pixel pair for each edge pixel detected.

The second approach simply marks either one of the two pixels as the edge pixel. One way to do this is to select the one with higher pixel intensity as the edge pixel because, statistically, the foreground is likely brighter than the background. This approach, in general, introduces an average error of half of a pixel, but it only needs to perform one reshading operation for each edge pixel. To simplify the implementation, all the results produced in this paper are based on this approach.

V. Constructing the Lookup Table

Since a local edge map may contain any number of edge contours, identifying them and selecting the most appropriate one for curve fitting can be computationally very expensive. The algorithm will also be very complex because it needs to handle many possible situations, making it extremely difficult to run efficiently. To overcome these limitations, we perform the curve-fitting process as a preprocessing task. During runtime, we only need to construct an index from the local edge map to access relevant information from a precomputed table of fitted curves. In this section, we show how curve fitting is done and how to construct the index table.

A. Curve Fitting

To fit a curve to some edge pixels, we need to link up the edge pixels. We have adopted the 8-connected approach and the chain-coding method [8] for linking the pixels. When tracing an edge contour, in most situations, an edge pixel P_n has two neighboring edge pixels P_{n-1} and P_{n+1}. P_{n-1} is the previous pixel which leads to P_n and P_{n+1} is the next pixel which continues from P_n. Sometimes, P_n may have more than two neighboring edge pixels. In this case, there would be at least two pixels to choose from to be the P_{n+1} pixel in the edge contour. We adapt the following two criteria for selecting the next edge pixel.

1) Always select a 4-connected neighbor in preference to a diagonal neighbor. This is because a 4-connected neighbor has a smaller Euclidean distance from P_n than the diagonal neighbor has.
2) If criteria 1) cannot resolve the situation, we select the one that continues from the previous edge direction.

Fig. 1 shows some examples of selecting the P_{n+1} pixel. The dark arrows represent previous selections while the hashed arrows represent next selections.

Another problem occurs at the beginning and at the end of an edge contour, where there are no previous and next edge pixels, respectively. In order to be able to reshade the first few edge pixels at the beginning of an edge contour, we mirror the chain codes of the first few edge pixels, but in the opposite direction. Fig. 2 shows two examples of such a replication of edge pixels. Here, we assume that five connected edge pixels are used for each fitting. Hence, we need to mirror two pixels in order to reshade the first edge pixel. A similar method is also used at the end of an edge contour. Since this method of extending an edge contour for antialiasing is only an approximation, in order not to overuse it, we detect all edge pixels whether they are higher or lower than the given threshold value as discussed in Section IV to prevent fragmenting the edges and we apply this method only when necessary.

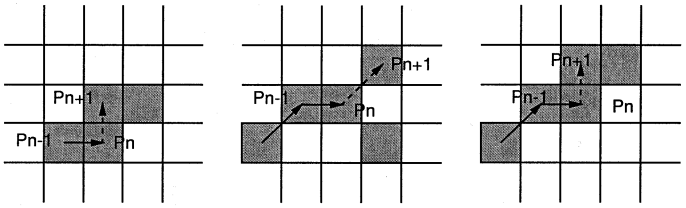


Fig. 1. Examples of edge contour tracing when there are multiple neighboring edge pixels.

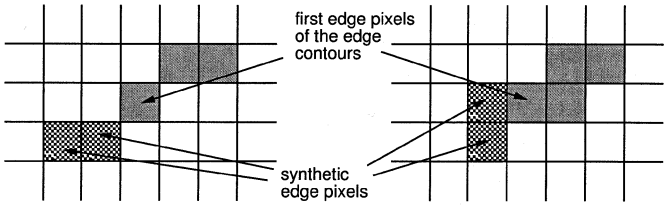


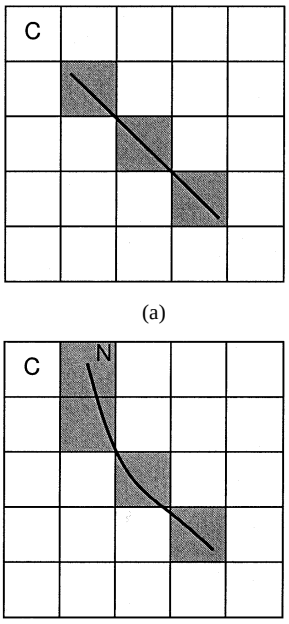
Fig. 2. Examples of synthesizing edge pixels at the beginning of an edge contour.

B. Forming the Indices

In our implementation, we use five connected edge pixels for curve fitting to reshade the edge pixel located at the center of the five pixels. To consider all possible local edge patterns produced by at most five connected edge pixels, we may need to look at a 5×5 pixel region, centered at the edge pixel to be reshaded. To decouple the curve-fitting process from the reshading process, we propose to construct a bitmask of this 5×5 pixel region, with each edge pixel having a value of "1" and each nonedge pixel having a value of "0." Using the contents of this bitmask as an index, we may determine the fitted curve, the alpha value and the locations to obtain the foreground and background colors from a precomputed table. Such decoupling is extremely important to the performance of the antialiasing process. Since the edge patterns can occur in many unexpected forms, determining the most suitable curve, the corresponding alpha value, and the locations to obtain the foreground and background colors of every possible pattern in real-time is extremely difficult and error prone.

However, if we simply consider the 5×5 bitmask as a 25-bit index, we may need to construct a table with 2^{25} entries storing the precomputed information. This will produce a table with 32M entries, which may be costly to put into a low-cost graphics accelerator. We observe the following two points.

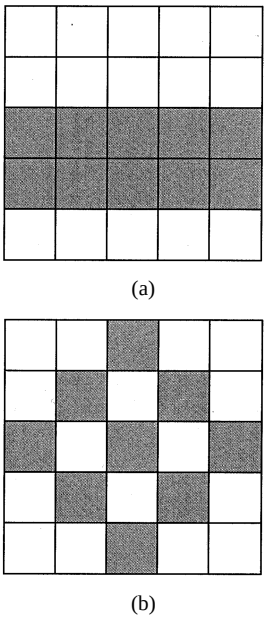
1) Since we only perform reshading on edge pixels, the center pixel of each 5×5 pixel region must be an edge pixel. Hence, we may not need to include the center pixel in the index.
2) Consider pixel C at the top left hand corner of Fig. 3. There are two possible types of edges that may be affected



(a)

(b)

Fig. 3. Two types of situation at the corner.



(a)

(b)

Fig. 4. Two examples that are difficult to perform curve fitting.

by the state of C. In Fig. 3(a), whether pixel C is an edge pixel, the fitted curve would still be a straight line. Therefore, the state of pixel C will not affect the resulting alpha value. Similarly, in Fig. 3(b), whether pixel C is an edge pixel, the fitted curve would still be the same since pixel N is a 4-connected neighbor. In other words, the state of this corner pixel does not affect the reshading result. This observation may be applied to the four corners of the 5×5 pixel region.

With the above observations, we may safely remove the center pixel and the four corner pixels from the index, without affecting the accuracy of the fitting. This results in an index of 20 bits and a table of 1M entries. If we use 8 bits to store the alpha value and 4 bits to store each of the foreground and the background pixel locations, each entry of the table will require 16 bits and the size of the table will then be 2M bytes. This memory space can easily be afforded by low-cost graphics systems.

C. Constructing the Index Table

Finally, we need to consider how to determine the fitted curves for the index table of 1M entries. For most edge patterns, we can easily determine the curves for fitting based on the two fitting rules defined in Section V-A. However, some complex edge patterns may be very difficult to determine appropriate curves for fitting automatically without human interactions. Fig. 4 shows two such examples.

To overcome this problem, we use a semi-automatic method to construct the table. A curve-fitting program will try to fill the table starting from the first entry. Whenever it encounters an edge pattern that it cannot handle, it signals the operator to indicate the appropriate curve manually. Sometimes, we may have situations that even the operator may not know the best way to fit a curve. For example, the edge pattern shown in Fig. 4(a) is likely due to a long thin horizontal object. It is difficult to do antialiasing that will actually help reduce the aliasing. In this situation, we may choose not to reshade the pixel. (We may indicate to the reshading process not to reshade this kind of edge patterns by setting both positions to obtain the foreground and background colors to zero in the corresponding entries of the index table.) This solution may minimize the potential error of the method in dealing with long thin objects.

Another problem that we may encounter is texture images. Texturing is a technique to map images to three–dimensional (3-D) models to improve realism, but without significantly increasing the modeling and rendering costs. Hence, it has become a very popular technique in real-time applications such as computer games. Unfortunately, texturing poses a potential problem to our method. Since our method decides if a pixel needs to be reshaded based on analyzing the rendered image alone, it may not be able to determine if an edge pixel is coming from the contents of a texture image or the discontinuity edges of a 3-D model. However, as we test our method on various texture images, we are surprised to see that the problem is not significant. This is because typical texture images are captured with a camera. These images are already low-pass filtered either by the camera or by the scanner. Hence, sharp edges would normally spread across two or more pixels and the contrast between two adjacent pixels is significantly reduced. As the Laplacian filter that we use only detects sharp edges appearing between two adjacent pixels, most edges appearing in a texture image will not be detected as discontinuity edges and hence will not be reshaded.

However, when the object to which a texture image is mapped moves away from the viewer, the texture image may be rendered to a small region (i.e., many texels will be mapped into a single pixel). In this situation, some edges may be detected as high-contrast edge pixels. However, this situation can be addressed in two possible ways. The first is to apply a low-pass filter to the texture image to generate a number of texture images of various resolutions. As the object is moving away from the

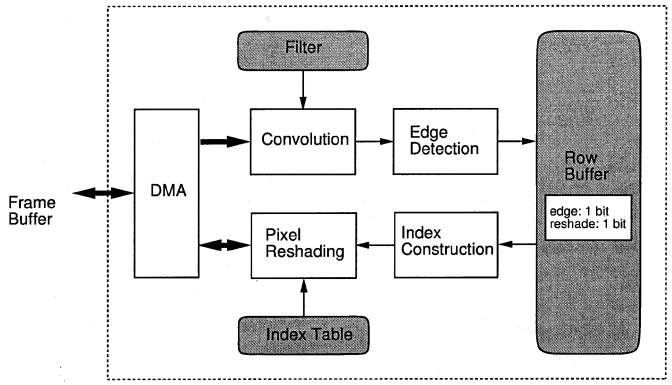


Fig. 5. Architecture of the postfiltering antialiasing method.

viewer, we may select a lower resolution image for rendering. (This technique, called *mipmapping*, is supported by OpenGL and is widely used for rendering texture images because it can speed up the rendering performance.) The second way to address this problem is through careful construction of the index table. When a texture image is displayed in a small region, the edge pixels become very dense. These dense edge patterns can be picked up as we construct the index table. Fig. 4(b) shows another example edge pattern. The cause of it is really application dependent. It may be due to a high-resolution texture image or some small objects. We may either construct different index tables for different types of applications or we may simply choose not to do reshading.

VI. Architecture

The block-level architecture of the proposed antialiasing method is shown in Fig. 5. It contains three memory blocks and four functional blocks. The usages of the memory blocks are explained as follows.

- *Filter*: It stores the contents of the *Laplacian Filter*, but an application may replace it with another filter more appropriate for the type of image being processed.
- *Index Table*: Each entry stores the precomputed curve-fitting information of a unique edge pattern. The information includes the α value, and the relative locations to obtain the foreground and background pixel values. Different tables constructed for different types of images may be dynamically loaded in here.
- *Row Buffer*: It is a two-bit shift buffer large enough to store more than five rows of the image edge values. The *edge bit* of each pixel is used to indicate if the pixel is an edge pixel. The *reshade bit* is used to indicate if the pixel should be reshaded, i.e., if it is a high-contrast edge pixel.

The functions of the four functional blocks are explained as follows.

1) The *DMA* block allows direct read/write accesses to the frame buffer.
2) The *Convolution* block convolves the filter function with the image stored in the frame buffer.
3) The *Edge Detection* block checks the filtered image data to determine whether a pixel is an edge pixel. The output is written to the *Row Buffer*.

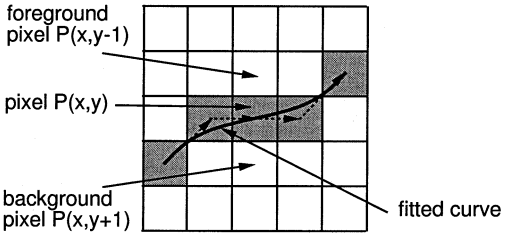


Fig. 6. Example of an edge segment.

4) The *Index Construction* block constructs an index from 5 $\times$ 5 edge bits read from the *Row Buffer*. The *Row Buffer* needs to store more than five rows of edge values so that an index can be constructed through a single read of the buffer. Concurrently, as the *Convolution* and the *Edge Detection* blocks are computing a new row of edge values, the oldest values can be shifted out of the *Row Buffer*.
5) The *Pixel Reshading* block computes the color value of each pixel with the *reshade bit* set.

VII. Results and Discussions

To show how we reshade a high-contrast edge pixel, we consider an example as shown in Fig. 6. The dotted arrows represent a traced edge segment. The solid line is a fitted curve for the center edge pixel, $P(x, y)$. We denote the region above the edge as foreground and the region below it as background. The index constructed for $P(x, y)$ is then

$$T(x, y) = 000\,00\,001\,0110\,10\,000\,000.$$

The relative location to obtain the color value of the foreground pixel is $(x_f[T(x, y)] = 0, y_f[T(x, y)] = -1)$, i.e., north of $P(x, y)$, and that of the background pixel is $(x_b[T(x, y)] = 0, y_b[T(x, y)] = 1)$, i.e., south of $P(x, y)$. These locations can be obtained by computing an orthogonal vector to the local orientation of the edge segment. The coverage value in this example is $\alpha_f[T(x, y)] = 0.5$. Hence, if the original intensity of $P(x, y)$ is $I_p(x, y)$, the new intensity value of it, $I'_p(x, y)$, can be recomputed as follows:

$$\begin{aligned} I'_p(x, y) &= 0.5 * I_p(x, y-1) + (1-0.5) * I_p(x, y+1) \\ &= 0.5 * (I_p(x, y-1) + I_p(x, y+1)). \end{aligned}$$

Note that here, only $T(x, y)$ and $I'_p(x, y)$ are computed at runtime. All the other values are computed in advance and stored in the index table.

To test our method, we have created an image with two small objects, a circle and a triangle. Fig. 7 shows the image with aliasing. Fig. 8 shows the same image but antialiased using supersampling. Fig. 9 is the output after applying the line-fitting antialiasing process described by Bloomenthal. We can see that while the filtered triangle is reasonably close to the one shown in Fig. 8, the circle appears polygonized. Fig. 10 is the output after applying our method. We can see that Figs. 8 and 10 are, in fact, very similar.

To demonstrate the output quality of our method, we apply the method to some video images produced by a typical mobile game console as shown in Fig. 11. Fig. 12 shows the same image when it is antialiased using supersampling. Fig. 13 shows

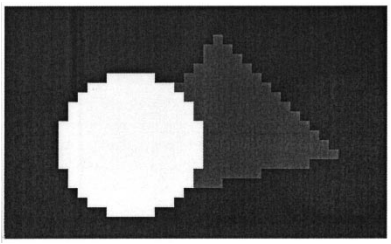


Fig. 7. Test image with two small objects.

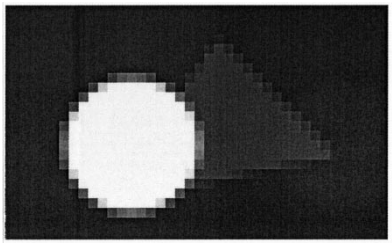


Fig. 8. Test image antialiased with supersampling.

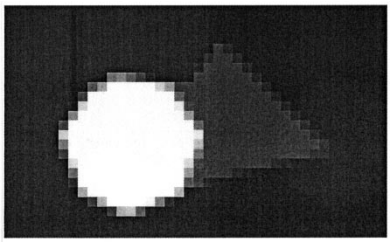


Fig. 9. Test image after the line-fitting antialiasing process.

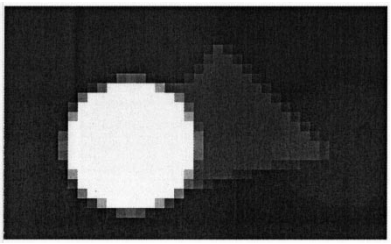


Fig. 10. Test image after the curve-fitting antialiasing process.

Fig. 11. Input image 1 with some 3-D objects.

Fig. 12. Image 1 antialiased using supersampling.

the extracted edges of Fig. 11. Fig. 14 is the output antialiased image of Fig. 11 using our method. Figs. 15–18 show another set of images similar to Figs. 11–14, but with texture. In general, the quality of the images after applying our method is much improved.

To demonstrate the performance of our method, we have measured the processing time on antialiasing a typical video game sequence, as shown in Fig. 19. The experiment was conducted on a PC with a Pentium III 866-MHz CPU. The program was implemented in Java and compiled as native code for execution. We can see that although the program was written in high-level language, its performance is still very high. This is due to the simplicity of the algorithm. We also observe that the antialiasing process only took about 10 ms to execute per frame. This process mainly involves an index construction step, a table lookup, and three color-blending operations (one for each channel) for each edge pixel needed to be reshaded. However, about 60 ms was spent on the edge-detection process. This process mainly involves one convolution operation and one discontinuity edge marking operation. The convolution operation is the most expensive one among all and takes more than 40 ms to execute (or about 67% of the processing time) per frame.

However, this convolution operation can be considered as a sequence of multiplication and addition steps, which can be implemented in hardware in a very efficient way. Most existing DSP chips have implemented this convolution operation as hardware logic by connecting a multiplier unit to an adder in the form of a loop, as shown in Fig. 20. (The i860 graphics

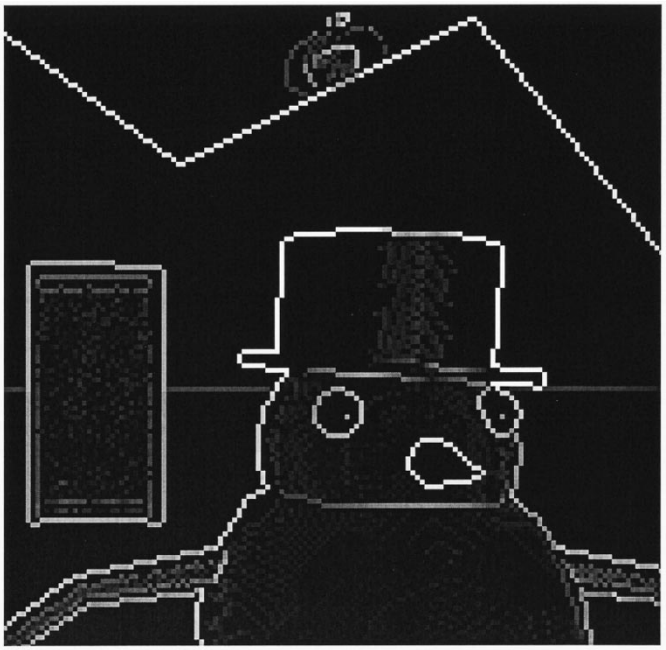


Fig. 13. Image 1 after the edge-detection process.

Fig. 14. Image 1 after the curve-fitting antialiasing process.

Fig. 15. Input image 2 with some 3-D objects.

Fig. 16. Image 2 antialiased using supersampling.

Fig. 17. Image 2 after the edge-detection process.

processor developed in the late 1980s also has similar hardware support [9].) With this configuration, one multiplication and one addition can be executed per cycle. Hence, it will only take five cycles to perform one single Laplician operation.

VIII. Evaluations

In summary, the proposed method has the following advantages. It is very simple to implement and extremely efficient in terms of computational cost. Basically, we only need to perform a Laplacian convolution operation and some comparison operations per pixel. If a pixel is found to be a high-contrast edge pixel, we perform a table lookup and a color-blending operation to reshade the pixel. In terms of memory cost, the main component for memory is the lookup table. As discussed earlier, the lookup table only requires 2M bytes of memory, which can easily be afforded by most low-cost systems. In addition, because the proposed method does not require to access the scene data, hardware implementation of it is extremely easy. It can be implemented as a chip and added to the end of the rendering pipeline, without any modification to the pipeline.

Fig. 18. Image 2 after the curve-fitting antialiasing process.

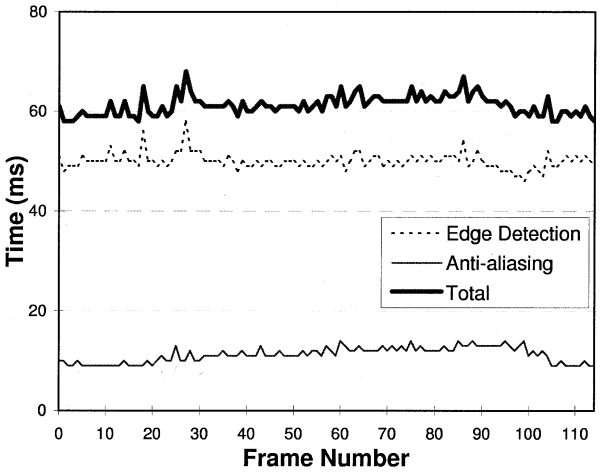


Fig. 19. Processing time for antialiasing a video sequence.

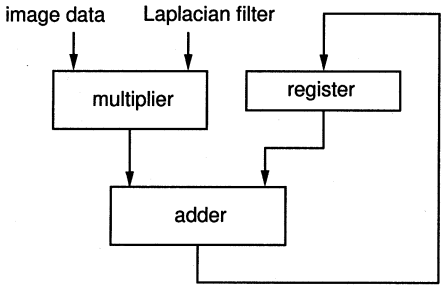


Fig. 20. Hardware support for the convolution operation.

Compared to Bloomenthal's method [3], the advantages of the proposed method are that the shapes of the filtered objects do not change according to which pixels are being selected as the first pixels for fitting, and there is no need to perform edge tracing in runtime. In addition, instead of supporting only line fitting, we can have both curve fitting as well as line fitting, depending on the local edge curvature.

Compared with the discontinuity edge overdraw method [17], our method does not require a lengthy preprocessing step and hence allows object deformation. In addition, our method, similar to Bloomenthal's method, is completely independent of the rendering method, while the overdraw method requires to have access to the scene data as well as the depth buffer. Furthermore, when two polyhedrons cross each other, discontinuity edges may be generated at the places where the polyhedrons intersect. As pointed out by [17], since these discontinuity edges are not in the geometry definitions, the overdraw method is not able to solve this problem. On the other hand, our method is based on detecting discontinuity edges appeared in the output image and hence can handle this problem. However, since the overdraw method obtains the edge lines for redraw directly from the scene data, the reshading result is more accurate while ours is only an approximation.

Compared to the A-buffer method [4] and its subsequent improvement methods [18], [19], our method is certainly much simpler. Although the A-buffer type of methods produces more accurate images, it requires runtime memory allocation and the amount of memory needed to render an image depends on the scene complexity. In addition, the number of fragments created for a pixel is not fixed nor known in advance. Furthermore, it requires all the fragments within each pixel to be dynamically sorted according to the depth value. This is why it is very complex and expensive to implement in hardware.

Compared to the accumulation buffer method [10], our method is much more efficient. The rendering performance of current graphics accelerators is mainly dictated by the polygon scan-conversion rate. Although the accumulation buffer method is already supported by most recent graphics accelerators, it does it by repeatedly scan-converting all the polygons as many times as the subpixel sampling rate. This essentially reduces the rendering performance of a given graphics accelerator by the subpixel sampling rate.

However, our method shares two common limitations with other postfiltering methods [3], [17]. First, since this kind of methods reshades the image after it is drawn, the error can be high if the environment contains a lot of small objects. Second, since it is not possible to determine the position of an edge within a pixel for an aliased image, there is an average of ± 0.5 pixel error on the reshaded edges. Although the discontinuity edge overdraw method accesses the scene data to redraw the edges, it also suffers from a similar problem. The redrawn lines will enlarge the size of the antialiased objects by $+0.5$ pixels on each side.

IX. Conclusion

In this paper, we have proposed using curve fitting (a general form of line fitting) as a postfiltering method for antialiasing. However, instead of identifying an appropriate curve to fit a particular edge contour in runtime, we identify all possible edge patterns and fit curves to each of these patterns in advance. An index table can then be precomputed to store all the fitting information. During runtime, we use the local edge patterns to form indices into this table to obtain information for reshading. The new method is very simple and efficient. We have demonstrated its performance and analytically compared its strengths and limitations with other methods. We have also shown some output images of the method.

A possible future work of this project may be to investigate how to make use of the scene data to assist in accurately locating

the discontinuity edges but without significantly increasing the preprocessing time.

ACKNOWLEDGMENT

The author would like to thank the anonymous reviewers for their useful comments and constructive suggestions to this paper, and Angus Siu for conducting some of the experiments.

REFERENCES

[1] K. Akeley, "Reality engine graphics," in *Proc. ACM SIGGRAPH'93*, Aug. 1993, pp. 109–116.
[2] K. Akeley and T. Jermoluk, "High-performance polygon rendering," in *Proc. ACM SIGGRAPH'88*, Aug. 1988, pp. 239–246.
[3] J. Bloomenthal, "Edge inference with applications to antialiasing," in *Proc. ACM SIGGRAPH'83*, July 1983, pp. 157–162.
[4] L. Carpenter, "The A-buffer, an antialiased hidden surface method," in *Proc. ACM SIGGRAPH'84*, July 1984, pp. 103–108.
[5] E. Catmull, "A subdivision algorithm for computer display of curved surfaces," Ph.D. dissertation, Comput. Sci. Dept., University of Utah, Salt Lake City, 1974.
[6] ——, "An analytic visible surface algorithm for independent pixel processing," in *Proc. ACM SIGGRAPH'84*, July 1984, pp. 109–115.
[7] F. Crow, "The aliasing problem in computer-generated shaded images," *Commun. ACM*, vol. 20, no. 11, pp. 799–805, Nov. 1977.
[8] H. Freeman, "On the encoding of arbitrary geometric configurations," *IRE Trans. Electron. Comput.*, vol. EC-10, pp. 260–269, June 1961.
[9] J. Grimes, L. Kohn, and R. Bharadhwaj, "The Intel i860 64-bit processor: A general-purpose CPU with 3D graphics capabilities," *IEEE Comput. Graph. Appl.*, vol. 9, no. 4, pp. 85–94, July 1989.
[10] P. Haeberli and K. Akeley, "The accumulation buffer: Hardware support for high-quality rendering," in *Proc. ACM SIGGRAPH'90*, Aug. 1990, pp. 309–318.
[11] M. Kelley, S. Winner, and K. Gould, "A scalable hardware render accelerator using a modified scanline algorithm," in *Proc. ACM SIGGRAPH'92*, July 1992, pp. 241–248.
[12] R. W. H. Lau and N. Wiseman, "Accurate image generation and interactive image editing with the A-buffer," in *Proc. Eurographics'92*, Sept. 1992, pp. 279–288.
[13] S. Lin, R. W. H. Lau, K. Hwang, X. Lin, and P. Cheung, "Adaptive parallel rendering on multiprocessors and workstation clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 12, pp. 241–258, Mar. 2001.
[14] S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: High-speed rendering using image composition," in *Proc. ACM SIGGRAPH'92*, July 1992, pp. 231–240.
[15] J. Montrym, D. Baum, D. Dignam, and C. Migdal, "InfiniteReality: A real-time graphics system," in *Proc. ACM SIGGRAPH'97*, Aug. 1997, pp. 293–301.
[16] A. Rosenfeld and A. Kak, "Edge detection," in *Digital Picture Processing*, 2nd ed., 1982, vol. 2, ch. 10.2, pp. 84–112.
[17] P. Sander, H. Hoppe, J. Snyder, and S. Gortler, "Discontinuity edge overdraw," in *Proc. ACM Symp. Interactive 3D Graphics*, 2001, pp. 167–174.
[18] A. Schilling, "A new simple and efficient antialiasing with subpixel masks," in *Proc. ACM SIGGRAPH'91*, 1991, pp. 133–141.
[19] A. Schilling and W. Straßer, "EXACT: Algorithm and hardware architecture for an IMproved A-buffer," in *Proc. ACM SIGGRAPH'93*, 1993, pp. 85–90.
[20] M. Waller, J. Ewins, M. White, and P. Lister, "Efficient coverage mask generation for antialiasing," *IEEE Comput. Graph. Appl.*, vol. 20, pp. 86–93, Nov. 2000.
[21] T. Whitted and D. Weimer, "A software test-bed for the development of 3-D raster graphics systems," in *Proc. ACM SIGGRAPH'81*, Aug. 1981, pp. 271–277.

Rynson W. H. Lau received the first-class B.Sc. (Hons.) degree in computer systems engineering from the University of Kent, Canterbury, U.K., in 1988 and the Ph.D. degree in computer graphics from the University of Cambridge, Cambridge, U.K, in 1992.

He is currently an Associate Professor at the City University of Hong Kong. Prior to joining the university in 1998, he taught at the Hong Kong Polytechnic University. From 1992 to 1993, he was with the University of York, York, U.K., working on a defense project on image processing. His research interests include computer graphics, virtual reality, and multimedia systems.