Image Analysis Applications and Computer Graphics, Chin et al. (Eds.), LNCS 1024, Springer-Verlag, pp. 205-214, Dec. 1995.

An Adaptive Supersampling Method

Rynson W. H. Lau

Computer Graphics and Media Laboratory, Department of Computing,
Hong Kong Polytechnic University, Hong Kong

Abstract. Original z-buffer method is a very efficient method for image generation. The limitation is that it introduces aliases into the output image. Although many different kinds of methods have been published to address this problem. most of them suffer from requiring a large memory space, demanding for high computational power, or having some other limitations. In this paper, we propose a simple anti-aliased method based on the supersampling method. However, instead of supersampling every pixel, we supersample edge pixels only. This method has the advantages of requiring less memory and less processing time than the traditional supersampling method. It is also less problematic than some other rendering methods in handling intersecting surfaces.

1 Introduction

Image generation based on the original z-buffer algorithm [5] requires only the colour value and the depth value of the closest object at each pixel. However, having these two values alone produces aliased images. This aliasing is the result of the point sampling nature of the z-buffer method. To solve this problem means that we need to solve the visibility problem in subpixel level. This requires the calculation of the visible area of each polygon at each pixel.

Existing hidden surface removal methods that calculate subpixel visibility can roughly be classified into *fixed-sized buffering (FSB) methods* in which the buffer used for image generation has a fixed memory size, and *variable-sized buffering (VSB) methods* in which the buffer does not have a fixed memory size.

Examples of FSB methods include supersampling z-buffer method [7] in which the scene is supersampled and then filtered down into the output resolution. The problems of this method are that it requires a lot of memory to store the supersampled image and a lot of processing time. The advantages, however, are that it is a simple extension of the z-buffer method and hence it can be implemented into hardware without too much addition effort [2, 11]. The RealityEngine [3] is also an hardware implementation of this kind although colour and depth are only sampled once per pixel to improve performance. To reduce memory usage, the RealityEngine uses the sparse mask method to reduce the number of samples per pixel. Four, eight or sixteen sample locations are chosen from an 8×8 coverage mask for anti-aliasing. Other FSB methods include [1] in which all polygons are required to clip against each other to produce a list of visible polygons before the scan-conversion process and [8] in which the scene is

sampled multiple times at different subpixel positions to produce multiple images which are combined to produce an output image. Both of these methods require less memory but considerable amount of processing time.

VSB methods can be further divided into *the span buffering methods* [6, 12, 14], which break polygons into horizontal spans and store in the buffer, and *the pixel buffering methods* [4, 10], which break polygons into pixel fragments and store in the buffer. A common feature of these methods is that they accumulate information of all visible objects in the buffer for hidden surface removal and anti-aliasing. In all these methods, memory usage depends on the complexity of the scene and there is no theoretical upper limit to it. Hence, they normally require run-time memory allocation. As such, real-time image generation using VSB methods is difficult. Hardware implementation of these methods are usually complex. [9] and [13] describe two rare hardware implementations of a span buffering method and of a pixel buffering method respectively.

In this paper, we describe a new scan-conversion method called the adaptive supersampling method. Although it is based on the supersampling method, it uses a variable-sized buffer called the adaptive supersampling buffer for adaptive supersampling. The new method tries to make a compromise between the supersampling method, which is easy to implement in hardware and can solve the surface intersection problem correctly but demands for a lot of memory and computation time, and the A-buffer method, which uses less memory and demands for less computation time but is difficult to implement into hardware and cannot solve the surface intersection problem correctly.

The outline of the paper is as follows. Section 2 describes the adaptive supersampling buffer. Section 3 discusses two implementation issues regarding to finding the depth increments and allocating run-time memory. Section 4 compares the strengths and limitations of the new method with three other major scan-conversion methods in terms of memory usage, performance and handling of surface intersection. Finally, Section 5 draws a conclusion of the paper.

2 Adaptive Supersampling Buffer

In the traditional supersampling method, memory is allocated to store the depth and colour values of each subpixel. Hence a lot of memory is needed and the actual memory usage is directly proportional to the subpixel resolution.

We have noticed that during the image generation process, there may be a lot of edge pixels created temporarily. These edge pixels may represent an edge of a polygon. However, when the connecting polygon is processed, an edge fragment from the former polygon may be merged with the edge fragment from the latter polygon and together they cover the whole pixel. Thus the edge pixel will become a non-edge pixel. In our experience, the maximum number of edge pixels for most of the complex images during the image generation process does not excess 20% the total number of pixels.

Because edge pixels contribute to only a small percentage of the total number of pixels in an image, our idea here is to supersample a polygon only when we

need to. This may result in a considerable amount of saving in both memory and processing time compared to the supersampling method.

To achieve this, when scan-converting a polygon, if the polygon covers the whole pixel, we sample the polygon once only. However, if the polygon partially covers the pixel, we perform a supersampling of the polygon within the pixel region. This demands for a buffer that can handle the information generated from either of the two sampling resolutions. Here, we apply a technique similar to the one used in the A-buffer [4]. A standard 2D buffer is used for normal z-buffer scan-conversion (one sample per pixel). When a polygon edge is encountered, a larger memory block is allocated to the pixel for storing the high resolution samples. However, unlike the A-buffer method, there is at most one memory block allocated to a pixel no matter how many polygon edges found in the pixel. In the A-buffer, there are two dimensions of uncertainty. The first one is the number of edge pixels in an image and the second one is the number of fragments in each edge pixels. The new method reduces it to one dimension – the number of edge pixels in an image. Because there is no need to traverse through a possible long list of fragments as in the A-buffer method, the new method greatly simplifies the algorithm and makes it easier to be implemented into hardware. In addition, the new method, like the supersampling method, can resolve surface intersection in subpixel level while the A-buffer method deals with the problem with an approximation method.

The new buffer uses two data structures as follows:

```
typedef struct {
    unsigned char r, g, b;
    unsigned char zflag;
    union {
        int z;
        PixelBlock *pblock;
    } zORpblock;
    short deltazx;  /* change of z in x axis */
    short deltazy;  /* change of z in y axis */
} Pixel;

typedef struct {
    int r[SUBPIXELS_Y][SUBPIXELS_X];
    int g[SUBPIXELS_Y][SUBPIXELS_X];
    int b[SUBPIXELS_Y][SUBPIXELS_X];
    int z[SUBPIXELS_Y][SUBPIXELS_X];
} PixelBlock;
```

Pixel is the basic element of a 2D pixel buffer. If a polygon completely covers a pixel, the **Pixel** element is used in a way similar to the traditional z-buffer method. However, when a polygon partially covers the pixel, the memory location

for storing **z** is used as a pointer instead. The flag **zflag** is set to false to indicate such a situation. **deltazx** and **deltazy** are used to store the the depth increments in the x and y directions respectively. These two values are calculated once for each polygon and are used to generate the depth values for each subpixel so that hidden surface removal may be performed in a subpixel level. **PixelBlock** is a run-time allocated memory block used to store the supersampled polygon information inside the pixel.

At start, the pixel buffer is initialised so that the contents of all **rgb** fields store the background colour. All **zflag**'s are set to true to indicate that **zORpblock** contains a depth value and all **z**'s are initialised with the maximum depth value. All **deltazx**'s and **deltazy**'s are set to zero.

Polygons are processed at random order and before scan-converting a polygon, we calculate its depth increments both in x and in y directions. When scan-converting the polygon, if a pixel is fully covered by it, the contents of **rgb** and **z** in the **Pixel** element are updated provided that the depth of the polygon at the pixel position is smaller than **z**. The pre-calculated depth increments of the polygon in the x and y directions are then stored to **deltazx** and **deltazy** respectively. If, instead, the pixel is partially covered by the polygon, a **PixelBlock** is requested and is linked to the corresponding **Pixel** element by setting **pblock** to point to the **PixelBlock**. **zflag** is then set to false to indicate that **zORpblock** is a pointer. All the **rgb** fields in the **PixelBlock** are initialised with the **rgb** values stored in the **Pixel** element and the **z**'s are initialised by interpolating the value of **z** stored in **Pixel** with **deltazx** and **deltazy**. Then the polygon fragment is supersampled to update the contents of **PixelBlock** based on the z-buffer method.

If a pixel with a **PixelBlock** is found fully covered by a polygon fragment with a smaller depth value, the information from the polygon is stored into the **Pixel** element and the **PixelBlock** is returned to the memory pool. However, if the pixel is partially covered by the polygon fragment, the polygon fragment is supersampled to update the contents of **PixelBlock** already stored in the pixel.

3 Implementation Issues

In this section, we discuss two implementation issues regarding to the calculation of the depth increments to be stored in **deltazx** and **deltazy** and the allocation of the **PixelBlock** memory.

3.1 Calculation of the Depth Increments

The depth increments, **deltazx** and **deltazy**, are the increments of **z** in x and in y directions respectively within the pixel region. From Figure 1, **deltazx** can be calculated as

$$\mathbf{deltazx} = \frac{Zx_{inc}}{X_{inc}} \quad .$$

However, in a three dimensional world, the polygon can have any orientation and calculating X_{inc}, Zx_{inc} and likewise Y_{inc}, Zy_{inc} are not straightforward.

Hence, instead, we calculate the depth increments from the polygon normal vector. After a polygon has been perspectively transformed, we calculate the unit normal vector of the transformed polygon as $\mathbf{N} = (dx, dy, dz)$. (This unit normal vector may already be available in some implementations of the rendering process. They are used for detecting back-surfaces.) With this normal vector, the depth increments can be calculated as

$$\mathbf{deltazx} = \frac{dx}{-dz} \quad and \quad \mathbf{deltazy} = \frac{dy}{-dz} \ .$$

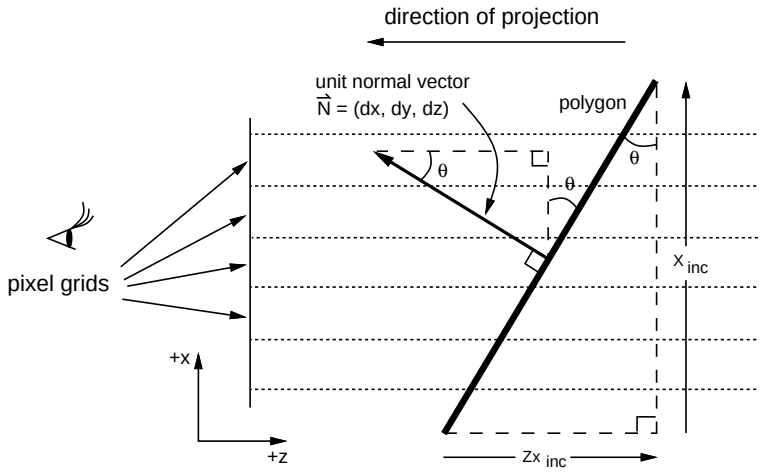

Fig. 1. Example of a perspective transformed polygon.

3.2 PixelBlock Memory

In order to reduce the frequency of calling the dynamic memory allocator provided by the compiler, which is usually not very efficient and tends to use up extra memory, we choose to pre-allocate a one dimensional array of **PixelBlock**'s in advance. Each **PixelBlock** in the array is made to point to the next, using one of the **z**'s, to form a list of free **PixelBlock**'s. When there is a request, the first **PixelBlock** in the list is returned and is put back to the head of the list when it is no longer needed. If the list is empty, we may then request the system for a new **PixelBlock** array.

4 Results and Comparisons

In this section, we compare our method with three other scan-conversion methods, the z-buffer method, the supersampling method and the A-buffer method, in terms of memory usage, processing time, and accuracy of determining surface intersection.

4.1 Memory Usage

Figure 2 compares the memory usage of the four scan-conversion methods against the number of polygon edges appearing in a pixel. The sizes of the data structures used in the calculation are based on our implementation of the methods — they are the minimum memory sizes needed to implement the methods. Except for the z-buffer method, the subpixel resolution of all the methods are 4×4. Each pixel of the z-buffer requires 8 bytes. Each pixel of the A-buffer requires 12 bytes but each fragment created will need 24 bytes of memory. (This calculation includes 4 extra bytes to store the second depth value for surface intersection detection.) However, the first polygon edge encountered will cause two fragments to be created, one for the polygon edge and the other for the background. Each pixel of the adaptive supersampling buffer requires 12 bytes but each **PixelBlock** created will need 112 bytes of memory. The supersampling z-buffer method, on the other hand, will need 112 bytes of memory for every pixel of the image.

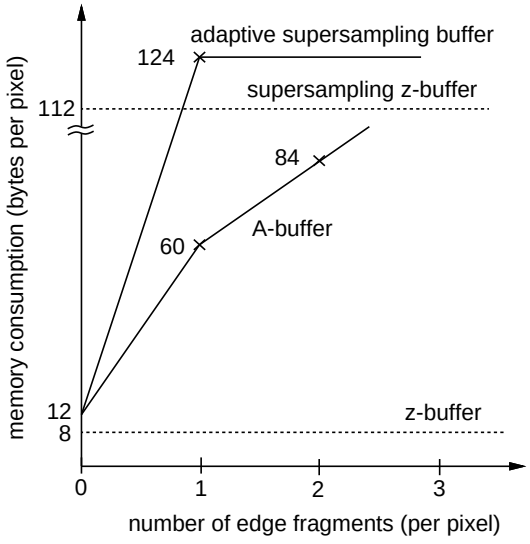

Fig. 2. Comparison of memory consumption against the number of edge fragments per pixel.

Figure 3 compares the memory usage of the four methods against the scene complexity. The size of the image used for the tests is 512×512 in resolution and the subpixel resolution is again 4×4. The scene complexity is proportional to both the total number of edge pixels and the total number of overlapping polygon edges in the image. The memory consumptions of the z-buffer method and the supersampling method are independent of the scene complexity. The memory consumption of the A-buffer method increases in a close to linear manner. Perviously, we mentioned that the maximum number of edge pixels for most of the complex images during the image generation process does not excess 20% of the total number of pixels. This is because as the scene complexity increases, more and more polygons tend to overlap each others. Hence, the memory consumption of our method will approach to a constant value as the complexity increases. This corresponds to a total memory size of 8.6M bytes.

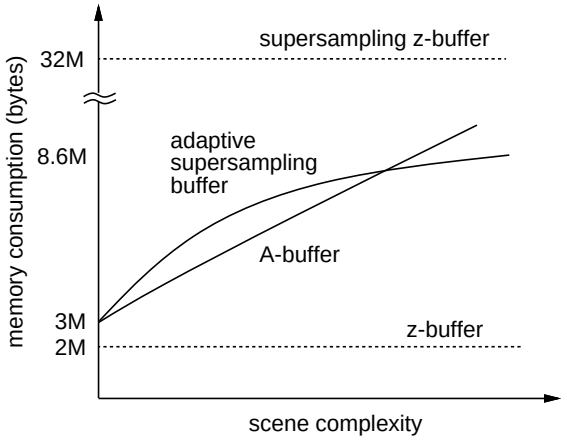

Fig. 3. Comparison of memory consumption against the scene complexity.

4.2 Performance

In our implementation of the four scan-conversion methods, all programs are written in C and in a similar way but differ only in the final scan-conversion parts. We tested the four scan-conversion methods with the image as shown in Figure 5. The following table shows the performance comparison of the methods and the results were taken on an SGI Indigo^2 machine with a 200MHz R4400 CPU. (The graphics accelerator in the machine was not used in the experiments.)

Rendering Methods	Times
Z-buffer method	1.74 sec.
Supersampling method	8.82 sec.
A-buffer method	2.05 sec.
Adaptive Supersampling method	1.96 sec.

From the results, the z-buffer method is obviously the fastest method of all. The supersampling method increases the rendering time dramatically. The new method has a slightly higher rendering time than the z-buffer method due to the extra computations in the edge pixels. However, it is interesting to see that the new method is also faster than the A-buffer method. After we have experienced with a few images with different complexity, we notice that when the scene complexity is low, the A-buffer method is faster and when the scene complexity is high, the new method becomes faster. The reason is that as the scene becomes more complex, more fragments will need to be processed in the A-buffer method and hence more floating-point operations will need to be executed to blend the colour values of the fragments, while in the new method, the number of edge pixels does not increase linearly with the increase in scene complexity and the merging of colour values within a **PixelBlock** involves only integer calculations.

In terms of memory consumption in rendering Figure 5, the A-buffer method uses a total of 87,075 fragments (equivalent to 2M bytes of memory) while the adaptive supersampling method uses a total of 22,081 **PixelBlock**'s (equivalent to 2.36M bytes of memory).

4.3 Handling of Surface Intersection

When surfaces intersect each other, we need to determine where they intersect in the subpixel level to prevent aliasing appearing along the line of intersection. In the supersampling method, the depth calculation and comparison are performed at each subpixel position. Hence, the surface intersection problem is solved automatically at the subpixel level. In the A-buffer method, the depth value is not available in subpixel level and an approximation method is suggested to detect the occurrence of surface intersection and to calculate where it happens. Two depth values representing the maximum and minimum depths of the fragment are stored for each polygon fragments. If two fragments overlap in depth, surface intersection is assumed to occur. The four depth values from the two fragments are then used to approximate the visibility of each of them. This method is problematic as has already been pointed out by Carpenter. One of the problems is that when two parallel surfaces are very close to each other and overlap each other in depth as shown in Figure 4, the A-buffer method will mistreat them as intersecting surfaces and blend them together. Although polygon B is not supposed to be visible but will appear as behind a somewhat semi-transparent polygon A. Similar to the supersampling method, our method can solve the surface intersection problem correctly in subpixel level.

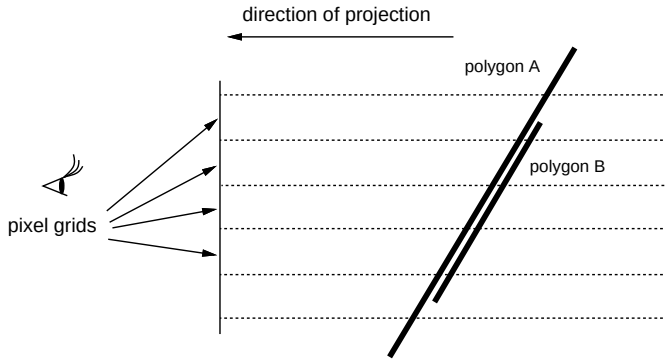

Fig. 4. Two parallel surfaces which are very close to each other.

5 Conclusion

In this paper, we have described a new buffer architecture for scan-conversion called the adaptive supersampling buffer. This method tries to make a compromise between the supersampling method, which is easy to implement in hardware and can solve the surface intersection problem correctly but demands for a lot of memory and computation time, and the A-buffer method, which uses less memory and demands for less computation time but is difficult to implement into hardware and cannot solve the surface intersection problem correctly.

References

1. Abram, G., Westover, L. and Whitted, T: "Efficient Alias-free Rendering Using Bit-masks and Look-up Tables". *Computer Graphics* **19**(3) (July 1985) 53–59.
2. Akeley, K., Jermoluk, T.: "High-Performance Polygon Rendering". *Computer Graphics* **22**(4) (Aug. 1988) 239–246.
3. Akeley, K.: "RealityEngine Graphics". *Computer Graphics* (Aug. 1993) 109–116.
4. Carpenter, L.: "The A-buffer, an Antialiased Hidden Surface Method". *Computer Graphics* **18**(3) (July 1984) 103–108.
5. Catmull, E.: *A Subdivision Algorithm for Computer Display of Curved Surfaces.* Ph.D. Dissertation, Computer Science Department, University of Utah (1974).
6. Catmull, E.: "A Hidden-Surface Algorithm with Anti-Aliasing". *Computer Graphics* **12**(3) (Aug. 1978) 6–11.
7. Crow, F.: "The Aliasing Problem in Computer-Generated Shaded Images". *Communication of the ACM* **20**(11) (Nov. 1977) 799–805.
8. Haeberli, P., Akeley, K.: "The Accumulation Buffer: Hardware Support for High-Quality Rendering". *Computer Graphics* **24**(4) (Aug. 1990) 309–318.
9. Kelley, M., Winner, S., Gould, K.: "A Scalable Hardware Render Accelerator using a Modified Scanline Algorithm". *Computer Graphics* **26**(2) (July 1992) 241–248.

10. Lau, W., Wiseman, N.: "Accurate Image Generation and Interactive Image Editing with the A-buffer". *Conference Proceedings of EuroGraphics '92* **II**(3) (Sept. 1992) 279–288.
11. Molnar, S., Eyles, J., Poulton, J.: "PixelFlow: High-Speed Rendering Using Image Composition". *Computer Graphics* **26**(2) (July 1992) 231–340.
12. Nakamae, E., Ishizaki, T., Nishita, T., Takita, S.: "Compositing 3D Images with Anti-aliasing and Various Shading Effects". *IEEE Computer Graphics & Applications* (Mar. 1989) 21–29.
13. Schilling, A., Straßer, W.: "EXACT: Algorithm and hardware Architecture for an Improved A-Buffer". *Computer Graphics* (Aug. 1993) 85–91.
14. Whitted, T., Weimer, D.: "A Software Test-Bed for the Development of 3-D Raster Graphics Systems". *Computer Graphics* **15**(3) (Aug. 1981) 271–277.

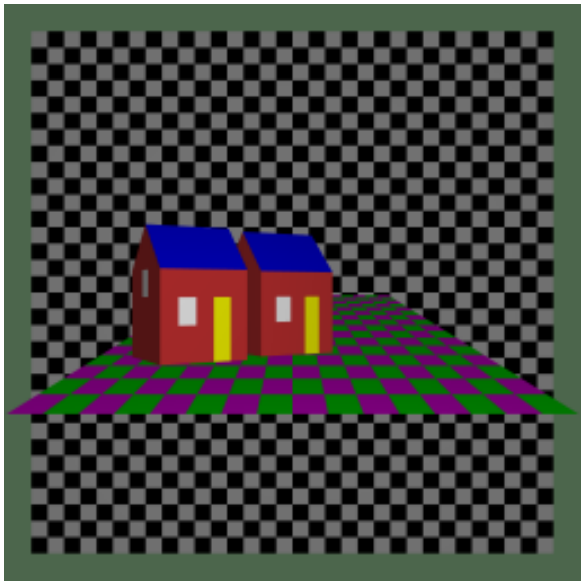

Fig. 5. Image generated using the adaptive supersampling method.

This article was processed using the LaTeX macro package with LLNCS style