Optimization and Implementation of Unstructured Grid Volume Rendering Algorithm

Jian Hu, Wenke Wang, Qianli Ma, Sikun Li
School of Computer, National University of Defense Technology

***Abstruct*-Visibility sorting is one of the key techniques in volume rendering of unstructured grids. The efficiency of sorting algorithm greatly affects the efficiency of volume rendering. Hardware-Assisted Visibility Sorting [1] (HAVS) is one of the most important volume rendering algorithms. This paper proposes an optimization algorithm of HAVS by keeping a sorted list which could decrease the comparing time in the fragment-level sorting process of k-buffer. The range of k is also analyzed by experiments. The results of experiment show that the proposed optimization algorithm obviously improves the efficiency of HAVS.**

Keywords -Volume rendering, Unstructured grids, Fragment, k-buffer

I. INTRODUCTION

Volume rendering is an important research field in science computing visualization and it can display the whole and details of 3D datasets. At present, volume rendering is considered to be the most important technique in 3D scalar sets visualization [2]. According to the difference of sorting, the visibility sorting techniques of unstructured grid volume rendering are divided into visibility sorting algorithms based on image space(A-Buffer[3] and R-buffer[4]), visibility sorting algorithms based on object space(NNS[5] and GATOR[6]) and visibility sorting algorithms based on both object space and image space(ZSWEEP[7] and HAVS[1]). Visibility sorting of fragment-level is one of the key technique for implementing volume rendering based on GPU, and the efficiency of sorting algorithm will greatly affect the efficiency of volume rendering. Visibility sorting algorithms based on image space must store all the fragments. The storing cost is too much and the rendering efficiency is low. Visibility sorting algorithms based on object space sorts triangle faces before rasterization. The speed of GPU is limited by the complexion of sorting in object space and the classification of triangles. Visibility sorting algorithms based on both image space and object space takes advantage of the processing ability of GPU in fragments-level, which shifts much of work from CPU to GPU and improve the rendering efficiency. Although ZSWEEP and HAVS are both visibility sorting algorithm based on both image space and object space, the sorting efficiency of ZSWEEP in object space is not high, which leads to too much burden for GPU, making the efficiency of ZSWEEP lower than HAVS. At present, HAVS is one of the most important unstructured grid volume rendering algorithm frameworks based on GPU[7]. However, fragments in *k*-buffer are out of order in the fragment process, and each time the algorithm needs 2*k*-1 comparisons to get 2 nearest fragments in depth, which lowers the rendering efficiency.

This paper proposes a visibility sorting algorithm to optimize the HAVS algorithm by keeping the *k*-buffer sorted. Each time the proposed algorithm just needs *k* comparison times to get 2 nearest fragments in depth, which obviously decreases comparison times. In addition, *k* will affect the precision and efficiency of the rendering result (the bigger *k* is, the more precise of the result and the lower of the efficiency are). The existing HAVS algorithm restricks *k* to 6, which limits the performance of algorithm. This paper researches on the range of *k* and obtains a *k* value that makes the result more precise and the efficiency higher. The experimental results show that the proposed algorithm obviously improves the efficiency of HAVS algorithm and obtains better performance for large dataset.

II. ANALYSIS OF HAVS ALGORITHM

A. *Brief description of HAVS*

HAVS is an unstructured grid volume rendering algorithm based on both object space and image space. The algorithm sorts the center of triangles using CPU in object space and sorts fragments using GPU in image space. Figure 1. shows the process of HAVS.

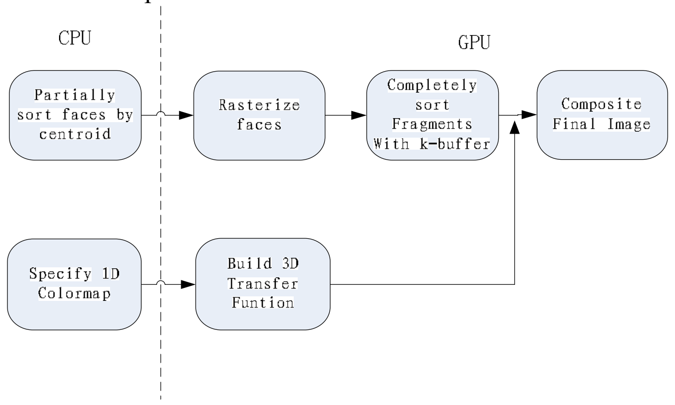
CPU
GPU
Partially sort faces by centroid
Rasterize faces
Completely sort Fragments With k-buffer
Composite Final Image
Specify 1D Colormap
Build 3D Transfer Funtion

Figure 1. Process of HAVS [1]

1) *Visibility Sorting in Object Space*

HAVS algorithm sorts triangle centroids using LSD radix sort and counting sort in object space. It uses the formula $f=f \times (-(f>>31)|0x80000000)$ to convert floating-point numbers into 32-bit unsigned integers and divide the integers into four 8-bit blocks. The counting sort algorithm is utilized to sort each block. Although the algorithm cost storage, the efficiency is high and can reach linear sorting time (*O(n+k)*, *n* is the length of the input array, *k* is the length of the computing array).

2) Visibility Sorting in Image Space

k-buffer is in charge of sorting fragments. Each pixel in the screen owns a buffer with length *k* and the buffer is called *k*-buffer. *k*-buffer stores *k* fragments ($f_1,f_2,\ldots,f_k$) for each pixel. Each fragment contains a value v_i and the distance between the viewport and the fragment. For front-to-back compositing, each time a new fragment f_{new} is coming, HAVS compares the distance d_{new} with the rest *k* fragments in the *k*-buffer ($d_1,d_2,\ldots,d_k$). Two fragments that have the smallest distance (denoted as f_i and f_j) are used to compute the color and opacity according to the 3D pre-integrated table. Then the color and opacity are composited to the framebuffer. Finally, the one that has smaller distance between f_i and f_j is deserted and the remaining fragments are written back to *k*-buffer. Because *k*-buffer can just consider *k* fragments once, if the fragments are highly out of order, the rendering result may be incorrect. When *k* is small, the comparison time is small, and thus the speed of the algorithm is high. However, the precision is low.

B. Time Cost of k-buffer

k-buffer is the key technique of HAVS, which uses a buffer with length *k* to store the fragments after rasterizing. If the number of fragments in buffer is already *k*, when a new fragment is coming, the algorithm composites two fragments nearest to the viewport, deserts the nearest one and writes back the rest *k* fragments to the *k*-buffer. The process of *k*-buffer is the process of selecting the two nearest fragments in depth. Each time it needs 2*k*-1 comparisons to gain the two nearest fragments because of the unsorted *k*-buffer. The details of the process are described as follows:

When the program starts, the number of fragments in *k*-buffer is smaller than *k*, f_{new} is directly inserted into *k*-buffer.

When the number of fragments in *k*-buffer is *k*, firstly, select a nearest fragment f_i, and then select the second nearest fragment f_j in the rest fragments in the buffer.

Use (v_m, v_n) and d_n-d_m to get the color and opacity from 3D preintegrated table. Composite the resulting color and opacity to the framebuffer.

Desert the one that has the smaller distance between f_i and f_j. Write back the remaining fragments to *k*-buffer.

Suppose the number of pixels in screen is $m\times n$ and the number of fragments in the *j*th pixel is s_j. The total comparison time required is analyzed as follows:

When the number of fragments in buffer is smaller than *k*, fragments are directly inserted into *k*-buffer, and thus the comparison time is 0.

When the number of fragments in buffer is *k*, firstly, select the nearest fragment from the *k*+1 fragments, the comparison time is *k*. Then, select the second nearest fragment from the rest *k* fragments, and the comparison time is *k*-1. There are s_j-*k* fragments need to compare, so the total comparison time is:

$$t_1 = (k + k - 1)\times(s_j - k)$$

When there only left no more than *k* fragments, HAVS needs to select the two nearest fragments. When there are *l* fragments, it needs *l*-1+*l*-2 comparisons. The total comparison time is:

$$t_2 = \sum_{l=2}^{k}(2l-3)$$

For all the $m\times n$ pixels in screen, the total comparisons are:

$$N_1 = \sum_{j=1}^{m\times n}(t_1+t_2) = \sum_{j=1}^{m\times n}\left(\sum_{l=2}^{k}(2l-3) + (k+k-1)\times(s_j - k)\right)$$

The average comparisons of each pixels is:

$$e_1 = \frac{N_1}{m\times n} = \frac{\sum_{j=1}^{m\times n}\left(\sum_{l=2}^{k}(2l-3) + (k+k-1)\times(s_j - k)\right)}{m\times n}$$

From the formula above, the average comparisons of each pixel increase drastically with the number of fragments increasing. Therefore, it is necessary to optimize *k*-buffer algorithm to decrease the comparisons for efficient rendering.

III. . Optimization of HAVS

As analyzed above, each time the algorithm needs 2*k*-1 comparisons to obtain the two nearest fragments because of the unsorted *k*-buffer. Although *k* is small, the number of fragments and pixels is large, the number of comparisons will also be very large. For this problem, we sort the *k*-buffer in ascending order of the distance of each fragment. Therefore, each time we select the first fragment in *k*-buffer as one of the nearest fragments and select the nearest fragment in the rest *k* fragments. Finally, the smaller one between the two fragments is selected and deserted. The comparisons decrease to *k* for each time. The details are as follows:

- When the number of fragments in k-buffer is smaller than k, the new fragment compares with the fragments already in the buffer and is inserted into the right position.
- When the number of fragments in k-buffer is k, select the first fragment f1 in the buffer as one of the nearest fragments.
- Select the nearest fragment fi in the rest k fragments.
- Compare f1 and fi, get smaller one fj
- Use (vi,v1) and the difference of distance to get the color and opacity from 3D preintegrated table. Composite the resulting color and opacity to the framebuffer.
- desert fj and write the remaining k fragments back to k-buffer.

The total comparisons required are analyzed as follows:

When the number of fragments in the buffer is smaller than *k*, the new fragment should be inserted into the buffer in order. Fot the *l*th insertion, the comparisons are *l*-1. So the total comparisons are $t_1 = \sum_{l=1}^{k-1} l$.

When the number of fragments in the buffer is k, firstly we select the first fragment in the buffer, which needs no comparing. And then select the nearest fragment in the rest k fragments, which needs k-1 comparisons. Finally, we select the smaller one from the two nearest fragments which needs 1 comparison. There are s_1-k fragments need to be compared, so the total comparisons are:

$$t_2 = (k - 1 + 1) \times (s_j - k)$$

When the number of fragments in the buffer is only k in the end. We only need to select the first 2 fragments in order, which needs no comparison. Therefore, the total comparisons are:

$$N_2 = \sum_{j=1}^{m\times n} (t_1 + t_2) = \sum_{j=1}^{m\times n} \left(\sum_{l=1}^{k-1} l + (1 + k - 1) \times (s_j - k)\right)$$

The average comparisons are:

$$e_2 = \frac{N_2}{m \times n} = \frac{\sum_{j=1}^{m\times n} \left(\sum_{l=1}^{k-1} l + k \times (s_j - k)\right)}{m \times n}$$

The difference of the average comparisons between the two algorithms are given as follows:

$$\Delta e = e_1 - e_2 = \frac{\sum_{j=1}^{m\times n} \left((k-1) \times (s_j - \frac{k}{2} - 1)\right)}{m \times n}$$

The improved efficiency is

$$\frac{\Delta e}{e_1} = \frac{\sum_{j=1}^{m\times n} \left((k-1) \times (s_j - \frac{k}{2} - 1)\right)}{\sum_{j=1}^{m\times n} \left(\sum_{l=2}^{k} (2l-3) + (k + k - 1) \times (s_j - k)\right)} = \frac{1}{2} - \frac{m \times n}{4\sum_{j=1}^{m\times n} s_j - 2(m \times n)(k+1)}$$

As the dataset increases, the later term of the formula will be smaller and the improved efficiency will be larger, and thus the optimization will be more notable.

IV. Experimental Results

k has great impact on the efficiency and precision of HAVS, the bigger k is, the more precise the result and the lower the efficiency are. The existing HAVS algorithm restricks k to 6, which limits the performance of the algorithm. Our experiments tested the value 6,8,10 and 12 for k. For each value of k, we tested 50 times and obtained the average time and the optimization efficiency. Figure 2-5 show the comparison of algorithm efficiency and time when k is 6,8,10 and 12 respectively (red color represents the time of HAVS and green color represents the time of the optimization algorithm). Figure 2 and 3 are the test results of pressure of NASA_yf17(97104 vertexes, 528915 cells). Figure 4 and 5 are the test results of the pressure grad of forward step shock flows from wind tunnel (278861 vertexes, 1500000 cells).

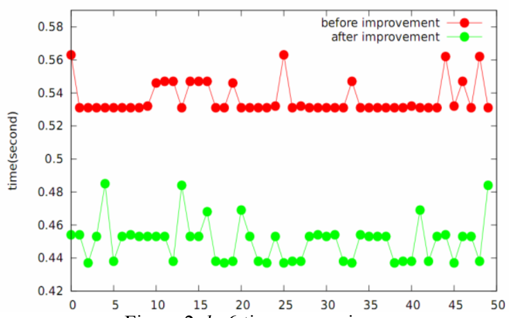

Figure 2. k=6 time comparison

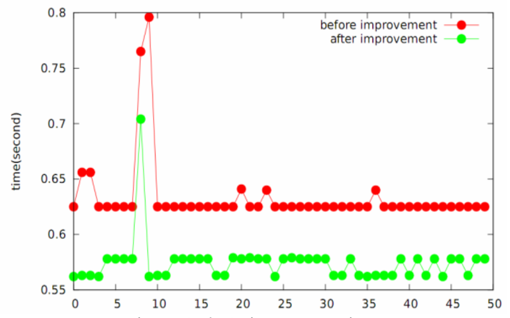

Figure 3. k=8 time comparison

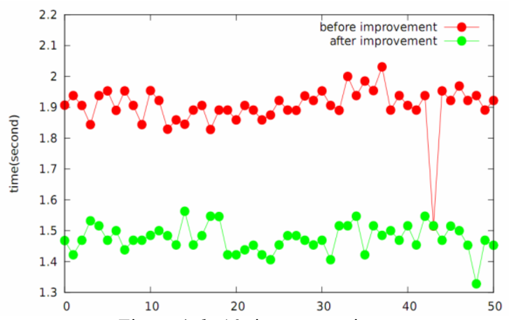

Figure 4. k=10 time comparison

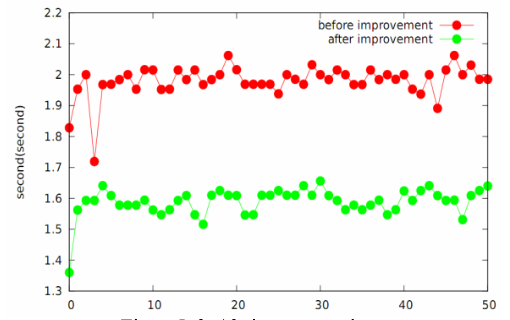

Figure5. k=12 time comparison

Our experiments are tested on a PC with Intel Pentium Dual 1.60GHz Processor(1G RAM) and NVIDIA GeForce9800 GTX with 512M RAM. Table 1 shows the rendering time of HAVS and optimization algorithm for 6,8,10 and 12 of k. It is concluded that our optimization obviously reduced the rendering time. Figure 6 and 7 display the images rendered by our optimization algorithm for datasets above. Pressure(or Pressure grads) is relatively bigger in red area, and is relatively smaller in green area.

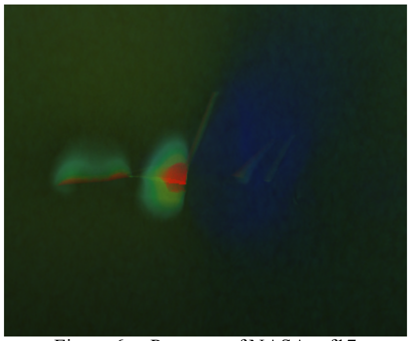
Figure 6. Pressure of NASA_yf17

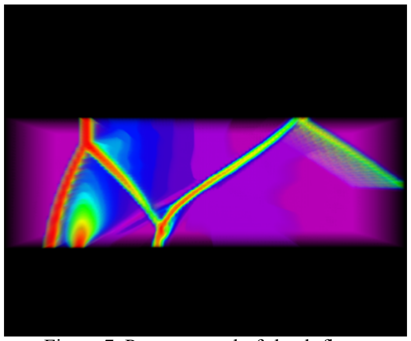
Figure 7. Pressure grad of shock flows

V. CONCLUSIONS

At present, HAVS is one of the most important unstructured grid volume rendering algorithm frameworks based on GPU. This paper proposes an optimization algorithm against the low efficiency of sorting in *k*-buffer. We also test different *k* values and extend the range of *k*, which will increase the precision of the final image. The experimental results show that our optimization algorithm obviously improves the efficiency of HAVS.

ACKNOWLEDGMENT

This work is supported by the National Basic Research Program (No. 2009CB723803).

REFERENCES

[1] Steven P. Callahan; Milan Ikits, Joa ˜o L.D. Comba; and Claudio T. Silva, Hardware-Assisted Visibility Sorting for Unstructured Volume Rendering, IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS,2005
[2] Tang Ze-sheng，Visualization of 3D Data Sets[M] Beijing:Tsinghua University Press 1999.12.
[3] L. Carpenter, "The A-Buffer, an Antialiased Hidden Surface Method," Computer Graphics (Proc. SIGGRAPH 84), vol. 18, no. 3, July 1984.pp. 103-108
[4] C. Wittenbrink, "R-Buffer: A Pointerless A-Buffer Hardware Architecture," Proc, ACM SIGGRAPH/Eurographics Workshop Graphics Hardware, pp. 73-80, 2001.
[5] M. Newell; R. Newell; and T. Sancha, "A Solution to the Hidden Surface Problem," Proc. ACM Ann. Conf., 1972,pp. 443-450.
[6] B. Wylie, K. Moreland; L.A. Fisk; and P. Crossno; Tetrahedra Projection Using Vertex Shaders,Proc. IEEE/ACM Symp. Volume Graphics and Visualization,2002,pp. 7-12
[7] R. Farias; J. Mitchell; and C.T. Silva, ZSWEEP: An Efficient and Exact Projection Algorithm for Unstructured Volume Rendering,Proc. IEEE Volume Visualization and Graphics Symp., 2000,pp. 91-99
[8] Silva; C. T., Comba; J. L. D., Callahan; S.P., Bernardon, F. F.. A survey of gpu-based volume rendering of unstructured grids[J]. Brazilian Journal of Theoretic and Applied Computing, 2005, 12(2): 9–29.

TABLE I. COMPARISON OF HAVS AND THE PROPOSED ALGORITHM

	k=6 HAVS		k=6 Optimization		
Dataset	*k*-buffer Process Time	Total Render Time	*k*-buffer Process Time	Total Render Time	Improved efficiency
NASA_yf17	0.53595s	0.73655s	0.4519s	0.6525s	11.41%
	k=8 HAVS		k=8 Optimization		
Dataset	*k*-buffer Process Time	Total Render Time	*k*-buffer Process Time	Total Render Time	Improved efficiency
NASA_yf17	0.6265s	0.8311s	0.56855s	0.77495s	6.97%
	k=10 HAVS		k=10 Optimization		
Dataset	*k*-buffer Process Time	Total Render Time	*k*-buffer Process Time	Total Render Time	Improved efficiency
Shock flows	1.90356s	2.59302s	1.4758s	2.16497s	16.50%
	k=12 HAVS		k=12 Optimization		
Dataset	*k*-buffer Process Time	Total Render Time	*k*-buffer Process Time	Total Render Time	Improved efficiency
Shock flows	1.97894s	2.63329s	1.58748s	2.24899s	14.80%