使用圖形硬體之容積視覺化平台
A Platform for Volume Visualization using Graphics Hardware

廖宏祥[1,2] 何丹期[2] 李柏穎[1] 張宏生[1]
Horng-Shyang Liao Tan-Chi Ho Po-Ying Li Charlie H. Chang

[1] Visualization and Interactive Media Laboratory,
IT and Visualization Division,
National Center for High-performance Computing

[2] Computer Graphics and Geometric Modeling Laboratory,
Department of Computer Science and Information Engineering,
National Chiao Tung University

E-mail: hsliao@nchc.org.tw, danki@csie.nctu.edu.tw,
bylee@nchc.org.tw, charlie.chang@nchc.org.tw
Address: 7, R&D 6th Rd., Science-Based Industrial Park, Hsinchu 300, Taiwan, R.O.C.
Tel: +886-3-5776085 Ext. 285

摘要

容積顯像法是一個相當重要的科學視算方法。它被廣泛的應用在醫療、生物資訊以及科學模擬等顯像。從2004年開始，我們持續開發容積顯像法到自製的三維虛擬實境顯像引擎中[12]。在這篇論文中，我們展示自行開發的容積顯像平台，它整合了觀視點切片、預先積分法、複合容積顯像、時間變異容積顯像與轉換函數之圖形化操作介面。此平台成功達到高顯像品質、複合容積顯像、時間變異容積顯像、以及轉換函數即時調整之需求。同時，我們保持實作之可擴充性，並且盡可能的使用圖學硬體加速。

ABSTRACT

Volume Rendering is an important scientific visualization method that is used widely in medical data, bioinformatic data, and scientific simulation results. In 2004, we started to develop volume rendering feature for our 3D VR Engine [12]. In this paper, we showed our work on creating this volume visualization platform, which integrated view-align slicing, pre-integration, multi-volumes rendering, time-varying volume rendering, and a graphics user interface for transfer function. Our platform met the needs for high quality volume rendering, multi-volumes, time-varying volume display, and had real-time adjustment for transfer function. Meanwhile, we maintained its extensibility and used graphics hardware functions when possible.

關鍵詞： 容積視覺化、預先積分法、複合容積、時間變異容積、轉換函數。

Keywords: Volume Visualization, Pre-integration, Multi-Volumes, Time-Varying Volume, Transfer Function.

1 Introduction

Volume rendering [4, 10] is a favorable presentation for 3D image data, such as medical images, bio-informatic data, and some simulation results. To provide a useful volume rendering display system, other than the integrity of final visualization images, real-time interaction is desired for better users perception. Moreover, capabilities of multi-volumes, time-varying volume data display, and feature detection are in constant demands. In 2004, we developed an axis-align, texture-based volume rendering for 3D VR Engine [8]. But the lack of rendering quality, order of mixed multiple volumes, time-varying volume data control, and feature detection left much room for improvement. In this paper, we developed reformed algorithms for these problems and put them together into a volume visualization platform to fulfill the users demands.

For quality issue, we chose view-align instead of axis-align volume rendering and then combined pre-integration method that was proposed by Engel et al. [6]. View-align polygons would sample volume by tri-linear interpolation of graphics hardware's 3D texture support. Pre-integration would reduce

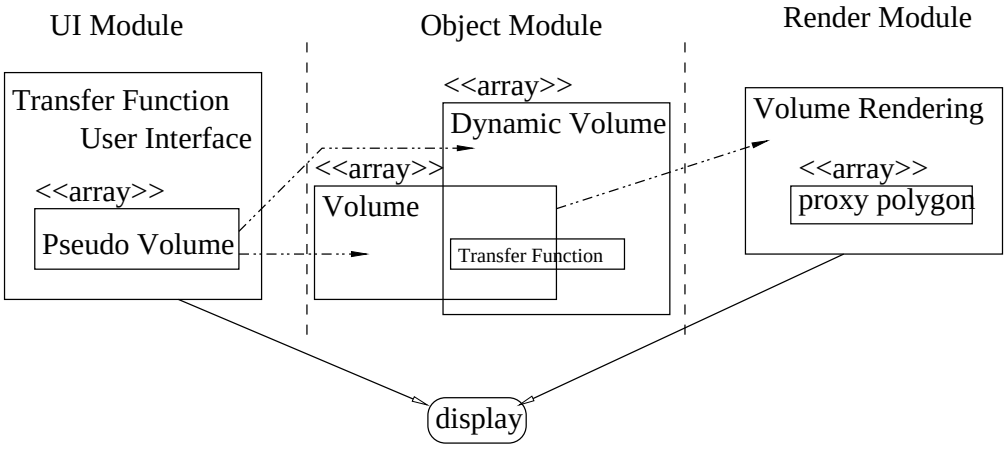


Figure 1: Structure of our volume rendering platform.

aliasing problem caused by the insufficient sampling slices. Engel implemented his pre-integration method with NVIDIA's register combiner OpenGL extension [2]. Our approach was using the more general OpenGL Shading Language (GLSL) [11]. The GLSL code was simpler and easier to maintain. For multiple volumes issue, we used slices sorting and texture switching to manage the display. View-align slices sorting was easy to get correct rendering order. For time-varying volume issue, we created a dynamic volume manager. For transfer function adjusting issue, we designed a control-points based GUI. There were histogram, color bar, and pre-integration table to assist user. Putting them all together, we built a texture-based volume rendering platform with hardware acceleration support.

2 Related Work

In recent years, graphics hardware becomes powerful and programmable. Therefore, making use of hardware acceleration becomes a reasonable trend to follow. There are increasingly more techniques of volume rendering to be implemented or accelerated by GPU. For example, Engel et al. hold a course in SIGGRAPH'02 to talk about this topic [5]. Therefore, we choose 3D texture [1, 3, 7] and GLSL [11] to be our fundamental tools.

According to the Nyquist Limit [14], we need more sampling slices to achieve high accuracy in direct volume rendering image. And the trade off is the performance slowdown. In 2001, Engel et al. proposed a pre-integration method to calculate the integration results between slices [6]. The method calculated slabs instead of slices and it could be implemented by the programmable graphics hardware.

Because a good transfer function is hard to find, Christopher R. Johnson lists the "feature detection" as one of the top scientific visualization research problems [9]. In 2001, a contest about transfer function searching was held [13]. They compared four different methods: trial and error, data-centric without data model, data-centric with data model, and image-centric using organized sampling. In the opinion of judge, Bill Lorensen, it was not good to have too much or too little human interaction. Semi-automatic method to assist user was much better. Following this comment, we decide to design our own control-points based GUI with assistant information. Current version is a easy-to-use GUI without automatic feature detection. We will extend it to support semi-automatic feature detection in next version.

3 Hardware Accelerated Volume Rendering

Following the organization of our 3D VR Engine [12], we divide our volume visualization platform into three major parts: data manager, rendering, and user interface. We also design a structure as **Fig. 1**.

Data manager is placed in object module. It has three objects: dynamic volume object, volume object, and transfer function object. Dynamic volume object contains volumes and manages volumes by time sequence (**Sec. 3.4**). Volume object manages volume data and its position and orientation. Both of them contain transfer function object. Transfer function object manages control points of transfer function and builds color table and pre-integration table (**Sec. 3.2** and **Sec. 3.5**).

Rendering is placed in rendering module. It has two objects: volume rendering object and proxy polygon object. Volume rendering object takes an array of volumes of this time step from dynamic volume object.

It will clip view-align slices volume by vol-

ume (**Sec. 3.1**). These slices will record in an array of proxy polygon objects and then are sorted (**Sec. 3.3**). Volume rendering object render these proxy polygons from back to front by pre-integration method (**Sec. 3.2**).

User interface is placed in UI module. It has two objects: transfer function user interface object and pseudo volume object. Transfer function user interface object provides a GUI to control transfer function (**Sec. 3.5**). Because transfer function can belong to dynamic volume object or volume object, we wrap them up by pseudo volume object and change pseudo volume from GUI.

3.1 View-Align Volume Rendering

In order to do view-align volume rendering, the first thing is to generate view-align triangles. View-align plane can be determined by view direction and distance between slice and camera. Therefore, the question becomes twofold.

1. How to find intersection points between view-align plane and volume?
2. How to triangulate these vertexes into non-overlap triangles that cover over all surface inside volume?

In order to find intersection points, we split eight points of volume's bounding box into two groups by view-align plane. Then, if any two vertexes pair has a boundary edge and places in different vertexes' group, there is a intersection point in this line segment. We apply line and plane intersection calculation to find intersection point.

The second question is more difficult. Based on our statistics, we find out that number of intersection points are no more than six points. Another important observation is that intersection points always form a convex polygon. These two facts can reduce many checkups. Based on number of intersection points, we deal with them case by case.

First, one intersection point case and two intersection points case are ignored and three intersection points case can link to a triangle directly.

In the case of four intersection points, like **Fig. 2(a)**, we do the following steps:

1. Calculate normal vectors from first point to others, $\overrightarrow{V_1V_2}$, $\overrightarrow{V_1V_3}$, and $\overrightarrow{V_1V_4}$.
2. Dot each pair of vectors, $\overrightarrow{V_1V_2} \cdot \overrightarrow{V_1V_3}$, $\overrightarrow{V_1V_2} \cdot \overrightarrow{V_1V_4}$, and $\overrightarrow{V_1V_3} \cdot \overrightarrow{V_1V_4}$.
3. Choose the largest two dot results, $\overrightarrow{V_1V_2} \cdot \overrightarrow{V_1V_3}$ and $\overrightarrow{V_1V_3} \cdot \overrightarrow{V_1V_4}$.
4. Create two triangles, $\triangle V_1V_2V_3$ and $\triangle V_1V_3V_4$.

These steps are based on convex polygon guarantee and larger dotted value smaller included angle.

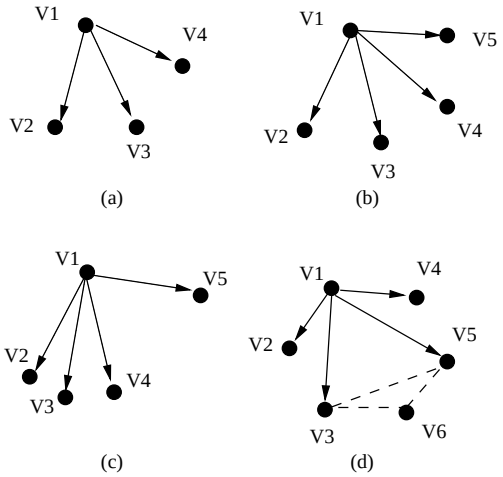


Figure 2: Triangles selection.

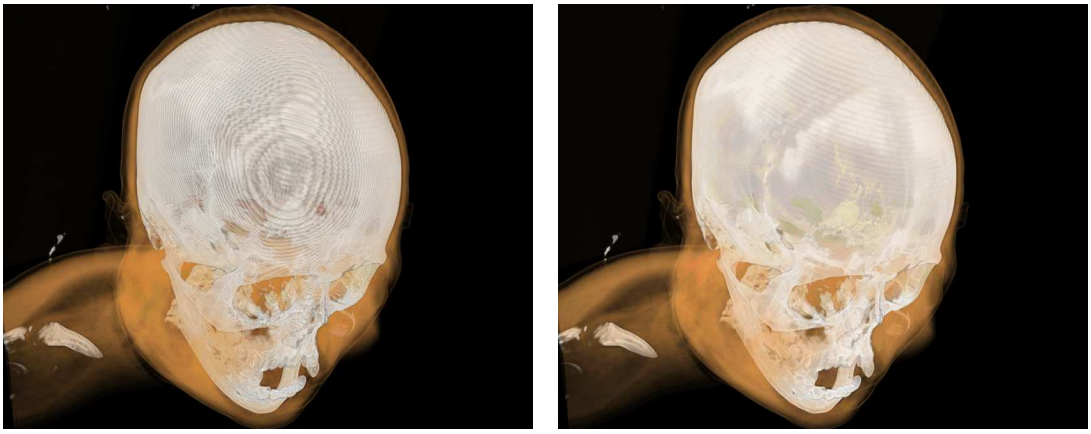
In the case of five intersection points, we do the same dotted methods and choose the largest three dotted values. For example, we will create three triangles $\triangle V_1V_2V_3$, $\triangle V_1V_3V_4$, and $\triangle V_1V_4V_5$ in **Fig. 2(b)**. But if the third largest dotted value forms a triangle that covers the previous two triangles, we ignore the third dotted value and choose the fourth dotted value. For example, we will skip $\overrightarrow{V_1V_2} \cdot \overrightarrow{V_1V_4}$ and choose $\overrightarrow{V_1V_4} \cdot \overrightarrow{V_1V_5}$ in **Fig. 2(c)**.

In the case of six intersection points, we can do the similar method again but it has too many special cases to avoid. Therefore, we use a reduction method to take care this case. We calculate normal vectors from the last point to others and dot each pair to choose the smallest value to form a triangle. It can reduce from six intersection points to five intersection points, and then we triangulate the remainder points by previous method of five intersection points. For example, we create $\triangle V_3V_5V_6$ in advance and then remove point V_6 to become a five intersection points situation in **Fig. 2(d)**.

After we generate correct view-align triangles and its texture coordinates, the next step is to build 3D volume texture and transfer function texture and then do the multi-texture mapping. The trilinear interpolation inside volume is done by graphics hardware automatically and transfer function mapping can be done in fragment shader by table look-up mechanism.

3.2 Pre-integration

Pre-integration algorithm integrates the composition result between slices. In other word, it samples volume by slabs instead of slices. This method reduces slicing alias and provide high quality volume rendering. **Fig. 3** is rendered by our implementation and provides a good proof. In this section, we will explain our implementation details: pre-integration table construction, slab calculation, and



(a) Without pre-integration.

(b) With pre-integration.

Figure 3: Pre-integration comparison.

table look-up. We will skip its fundamental theory and refer to the work by Engel et al. to elaborate the theory derivation [6].

First of all, we calculate pre-integration table. An example is shown in **Fig. 5**. The most right table is pre-integration table. Its horizontal axis is the values from front slice and its vertical axis is the values from back slice. RGB values in the table are integrated by Riemann Sum [15] and then normalized by its distance. Each alpha value is calculated by the following equation that is proposed by Engel et al. [6].

$$\alpha(s_f, s_b, d) \approx 1 - exp\left(-\frac{d}{s_b - s_f}(T(s_f) - T(s_b))\right)$$

with the integral function $T(s) := \int s0\tau(s)ds$. $\alpha(s_f, s_b, d)$ is target slab alpha value, s_f is value of front slice, s_b is value of back slice, and d is slab width.

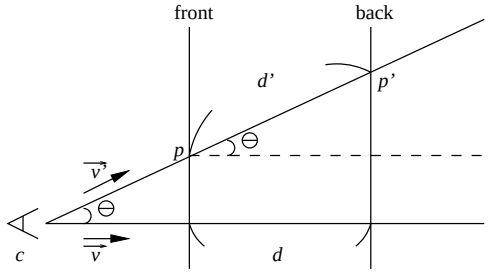


Figure 4: Calculate texture coordinate of back slice.

Next, we calculate texture coordinates of slab by vertex shader. Our targets are texture coordinates of back slice (**Fig. 4**). Vertex p and texture coordinates $TexCoord(p)$ of front slice are known information in vertex shader. We pass volume size $VolSize$, volume minimum corner $VolMin$, camera position c, unit view direction $\vec{v}$, and slab size d into vertex shader, too. We define vector $\vec{v'}$ as unit vector of camera to vertex of front slice. We calculate $\vec{v'} = normalize(p - c)$. If Θ is the angle between $\vec{v}$ and $\vec{v'}$ and d' is the distance between slices along the vector $\vec{v'}$, the following equation is established.

$$cos(\Theta) = d/d' = \vec{v} \cdot \vec{v'}/|\vec{v}||\vec{v'}|$$

Because v and v' are unit vectors,

$$d' = d/\vec{v} \cdot \vec{v'}$$

We define vertex of back slice as p' and texture coordinate of back slice as $TexCoord(p')$. We can derive the following results.

$$p' = p + d'\vec{v'}$$

$$TexCoord(p') = (p' - VolMin)/VolSize$$

After pre-integration table and texture coordinate are ready, the final step is texture lookup in fragment shader. We use multi-texture to bind volume as 3D texture and pre-integration table as 2D texture. First, we look up value in 3D texture by texture coordinates of front slice and back slice separately. We get two results from previous texture look-up and then use the two values as texture coordinates of pre-integration texture to get the final RGBA colors of the fragment in frame buffer. Alpha blending will complete the volume rendering after the previous fragment shader.

3.3 Multiple Volumes

Traditional texture-based multiple volumes rendering uses multiple textures to represent multiple volumes. This method has two major drawbacks.

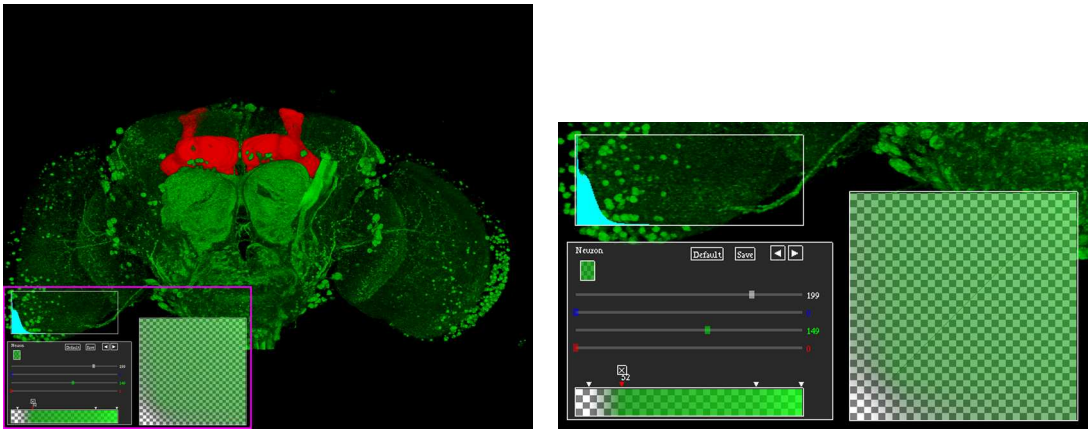


Figure 5: Transfer function GUI.

First, number of textures for one polygon are limited. Second, these volumes must be aligned.

In our multiple volumes situation, every volume has its own volume size, sampling slices, position, and orientation. Therefore, we cannot use the previous method. Our solution is based on slices sorting and texture switching concepts. Because our sampling slices are view-align slices, they can be sorted easily by distance between slice and camera. Four parameters are recorded for these slice polygons: volume index, distance from camera to slice, vertexes, and texture coordinates. Distance is sorting basis. Vertexes and texture coordinates are calculated by algorithm of **Sec. 3.1**. Because the two parameters are easier to be calculated volume by volume, they are calculated before sorting and rendered after sorting. Volume index is used to determine that this polygon belong to which volume. Our volume object has ID of texture object, transfer function, transform, and slab attributes to assist final rendering. After the sorting, all slices in the scene are rendered from back to front and multiple volumes are rendered in correct visual effect.

3.4 Time-Varying Volume

We design a dynamic volume class to handle time-varying volume. In dynamic volume class, it has three volume pointers, one for previous volume, one for current volume, and one for next volume. This class will maintain another thread to manage data. The original rendering thread renders the volume that is reported by data manager thread. Data manager thread loads volume and points it by next pointer. When next volume is loaded completely, next volume is changed to current volume and starts to load volume of next time step immediately. When current volume is ready, previous volume is freed immediately. If rendering thread asks volume pointer at this moment, data manager thread will report current volume pointer. When previous volume is freed completely, current volume is changed to previous volume and reset to be ready to take data from next volume. If rendering thread asks volume pointer at this moment, data manager thread will report previous volume pointer. This algorithm is not very efficient but it can render larger time-varying data then load total time-varying data. It is the reason for us to use this algorithm for time-varying volume.

3.5 Transfer Function User Interface

After we finish our transfer function implementation, we realize that we need a good transfer function GUI. Our first version of transfer function GUI is similar to the GUI that is used by Engel et al. [6]. We can assign RGBA values separately and immediately in this GUI. But, we have two problems when we use this kind of GUI. First, it is hard to give curves with specific signification. Second, we need more hints and guidance when users do trial-and-error. Therefore, we start to design our own transfer function GUI.

Fig. 5 illustrates the GUI of our transfer function editor. Our design concept is control-points based GUI. The main control window places in left-bottom. The top window is histogram result of current chosen volume. The right window is pre-integrated result of transfer function. Histogram window and pre-integration window provide assist information. Many features are inside lower statistic values, higher statistic values, and dramatic variant values. In control window, volume name and color of current chosen control point are shown. In top area of control window, default button will reload default transfer function from script, save button will save current transfer function into a file that is named by the volume's name, left arrow changes chosen volume to previous one, and right

arrow changes chosen volume to next one. The color bar in bottom of control window displays the transfer function. User can click on it to add control points. If user clicks on control point, control point will become red, density value, remove icon, and four color mapping bars will appear, and user can drag chosen control point to change source mapping density value. When control point is chosen, user can drag the four points on the four color bars to change target mapping color values or click the removed icon to remove this control point. When user changes any value of any control point, our program will do linear interpolation between two closest control points.

Our proposed transfer function GUI can provide a fundamental solution for the two issues that we mentioned previously. For more convenient usage, we will continue to extend it, such as semi-automatic detection, gradient histogram, and the like.

4 Experimental Results

We test our program on a PC with dual AMD Athlon MP 2800+ CPU, 3GB Register ECC DDR266 memory, and NVIDIA GeForce 6800 GT graphics card with 256MB graphics memory. Our program is developed in win32 platform and tested in Microsoft Windows 2000 Professional.

Fig. 6 is the rendering results. **Fig. 6(a)** and **Fig. 6(b)** are high resolution images of fruit fly brain from confocal microscopy. Each neuro-circuit needs to be recorded by a single volume. Therefore, they are displayed in a multiple volumes' environment. **Fig. 6(c)** is a CT human brain data. **Fig. 6(d)** is a CT lung cancer data and red parts inside the two pink squares are our experimental detection of cancer. **Fig. 6(e)** is a water simulation about "drop structure flow in an open-channeled hydraulic conduit." This simulation provides a time-varying data. We display it in a time-varying environment. **Fig. 6(f)** is a radar data of Herb typhoon. It is a hybrid rendering including typhoon volume and Taiwan terrain. All data type of these volumes is one 8-Bits channel. Their rendering performance is listed in **Table 1**.

5 Discussion

According to **Table 1**, our simple summary is larger data with lower rendering performance. But, there are three special cases. First, time-varying data are large but performance are not bad. This is because total time-variant volume data don't need to be rendered at the same time. It is only possible to render one time step volume at one frame. So, the rendering cost is limited to one time step volume rendering cost. Second, the bottleneck of hybrid rendering in **Fig. 6(f)** is dependent on the highest cost rendering in the scene. Our testing result is the best view of typhoon. In this case, our Level-Of-Detail (LOD) terrain chooses a very coarse terrain model. Most part of rendering cost come from typhoon's volume rendering. If we change observation viewpoint to the bottom of typhoon, LOD terrain chooses a more detailed terrain model. The bottleneck can come from terrain and FPS might drop to lower 30. Finally, the strangest result is that the FPS of **Fig. 6(c)** is lower than FPS of **Fig. 6(a)**. Volume resolution of **Fig. 6(c)** is lower than **Fig. 6(a)**. In our opinion, voxel size also acts as an important coefficient. Voxel size will decide the final volume size and shape. The different volume shape changes the cost of rasterization. Human brain data is more like a cube than fruit fly brain data. Therefore, it needs more pixels to calculate the final image.

6 Conclusion

We develop a simple and flexible volume rendering method, which can be run in current consumer PC and graphics hardware, for our 3D VR Engine [12]. We implement pre-integration algorithm [6] using GLSL to solve aliasing problem. Multiple volumes, time-varying volume, and an easy-to-use transfer function GUI also add up well to this platform.

In the future, high resolution volume data will be the next challenge to our rendering platform. We are trying some parallel or multi-resolution methods to overcome this issue. Another important work is semi-automatic feature detection as we have mentioned before.

Acknowledgements

We thank our team members: Dr. Ching-Yao Lin, Ming-Jing Li, Yan-Jen Su, and Chia-Yang Sun for their image processing toolkits (IMT) supporting and our resource supporters: Cherng-Yeu Shen and San-Liang Chu. We also thank our testing data providers: Dr. Chang-Huain Hsieh and Prof. Ann-Shyn Chiang for brain of fruit fly data, Chang Gung Memorial Hospital for CT data of human brain and lung cancer, Center for Space and Remote Sensing Research National Central University for terrain data, Hydrotech Research Institute National Taiwan University for typhoon's radar data, Lung-Cheng Lee for the simulation result of drop structure flow in an open-channeled hydraulic conduit.

Table 1: Performance.

Figure	Volume Resolution	Voxel Size	Time Step	FPS
Fig. 6(a)	512×512×128	1.8447265625×1.0078125×1.5	1	3.5
	512×512×128	1.8447265625×1.0078125×1.40625		
Fig. 6(b)	512×512×256	0.449841×0.449841×0.999365	1	0.5
	512×512×256	0.590412375×0.571960716796875×1.0		
	512×512× 32	0.68003609765625×0.54736633984375×2.5		
	512×512× 64	0.448055×0.406884×2.75992		
	512×512×128	0.40746962109375×0.491254×1.99489		
	512×256×256	1.220703125×1.28125×1.0		
	256×256× 64	2.44140625×1.28125×1.0		
	256×256×128	2.44140625×1.28125×1.0		
	256×256×256	2.44140625×1.26953125×1.0		
Fig. 6(c)	512×512×128	0.5742187×0.5742187×2.0	1	2.2
Fig. 6(d)	512×512×512	0.66015625×0.66015625×1.0	1	1.5
	512×512×256	0.66015625×0.66015625×1.0		
Fig. 6(e)	256×256×256	1.0×1.0×1.0	24	4.5
Fig. 6(f)	512×512× 16	1.5625×1.5625×0.75	1	30

References

[1] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Proceedings of the 1994 Symposium on Volume Visualization*, pages 91–98, 1994.

[2] NVIDIA Corporation. NVIDIA OpenGL Extension Specifications. http://developer.nvidia.com/object/nvidia_opengl_specs.html.

[3] T. J. Cullip and U. Neumann. Accelerating Volume Reconstruction with 3D Texture Hardware. Technical report, 1994.

[4] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume Rendering. In *Proceedings of Computer Graphics*, volume 22, pages 65–74, August 1988.

[5] K. Engel, M. Hadwiger, J. M. Kniss, and C. Rezk-Salama. High-Quality Volume Graphics on Consumer PC Hardware. *SIGGRAPH'02 course notes*, 42, 2002.

[6] K. Engel, M. Kraus, and T. Ertl. High-Quality Pre-integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, pages 9–16, 2001.

[7] A. V. Gelder and K. Kim. Direct Volume Rendering with Shading via Three-Dimensional Textures. In *Proceedings of the 1996 Symposium on Volume Visualization*, pages 23–31, 1996.

[8] T.-C. Ho, H.-S. Liao, P.-Y. Li, Charlie H. Chang, and S.-L. Chu. Dynamic Window Level Change of Texture Mapping Based Volume Rendering in NCHC's 3D VR Engine. In *Proceedings of 2004 Computer Graphics Workshop (CGW 2004)*, 2004.

[9] C. R. Johnson. Top Scientific Visualization Research Problems. *IEEE Computer Graphics and Applications*, 24(4):13–17, 2004.

[10] A. Kaufman. *Volume Visualization(IEEE Computer Society Press Tutorial)*. IEEE Computer Society, January 1991.

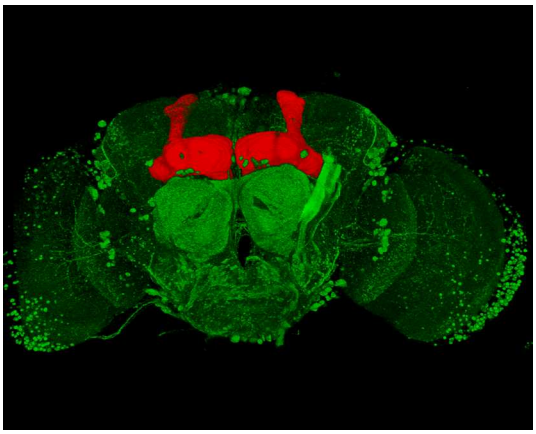
[11] J. Kessenich, D. Baldwin, and R. Rost. THE OpenGL Shading Language. http://www.opengl.org/documentation/oglsl.html.

[12] H.-S. Liao, P.-Y. Li, Charlie H. Chang, and S.-L. Chu. 3D VR Engine. In *Proceedings of 7th International Conference on High Performance Computing and Grid in Asia Pacific Region (HPC Asia 2004)*, 2004.

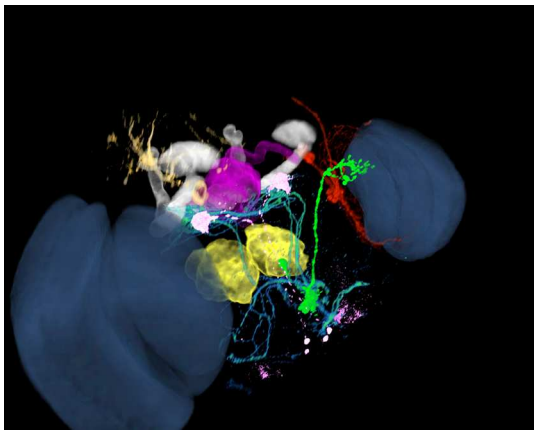
[13] H. Pfister, B. Lorensen, C. Bajaj, G. Kindlmann, W. Schroeder, L. S. Avila, K. Martin, R. Machiraju, and J. Lee. The Transfer Function Bake-Off. *IEEE Computer Graphics and Applications*, 21(3):16–22, 2001.

[14] E. W. Weisstein. Nyquist Frequency. From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/NyquistFrequency.html.

[15] E. W. Weisstein. Riemann Sum. From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/RiemannSum.html.

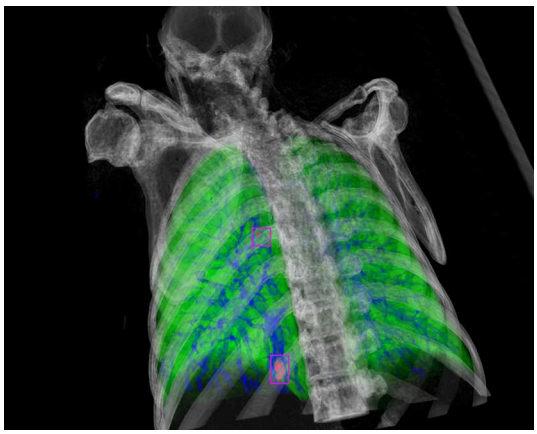


(a) Brain of fruit fly.

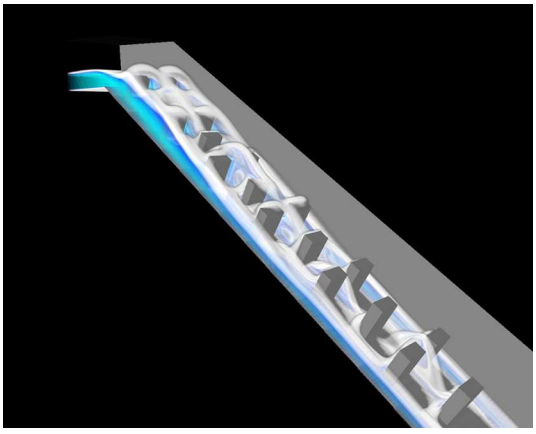


(b) Brain of fruit fly.

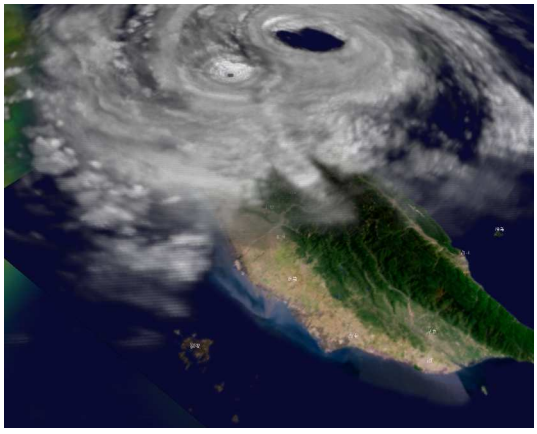
(c) Human brain.



(d) Lung Cancer.



(e) Drop structure flow in an open-channeled hydraulic conduit.



(f) Typhoon.

Figure 6: Volume rendering results.