國立交通大學

資訊科學與工程研究所

碩 士 論 文

共軛焦顯微鏡產生之果蠅腦影像之顯像系統

A Visualization System for Confocal Microscopic Image of Drosophila's Brain

研 究 生：林進錕

指導教授：荊宇泰　教授

中 華 民 國 九 十 七 年 九 月

共軛焦顯微鏡產生之果蠅腦影像之顯像系統

學生：林進錕　　　　　　　　　　　　　　指導教授：荊宇泰　博士

國立交通大學資訊科學與工程研究所

摘　　要

為了了解果蠅腦的功能，三維立體影像的腦圖譜和腦內結構位置的建立，並且進一步了解神經網路如何的連結是非常重要的。針對共軛焦顯微鏡產生之果蠅腦資料，我們建立了一個以貼圖為基礎的立體資料描繪系統。為了比較不同果蠅腦的結構，我們的系統可同時描繪多組立體資料。打光和陰影效果及預先積分之立體資料描繪法更進一步的在我們系統中被支援以達到更精緻的描繪結果。經由神經追蹤產生之幾何資訊也可以在我們的系統中被完美的結合。

A Visualization System for Confocal Microscopic Image of *Drosophila's* Brain

Student: Jin-Kuen Lin Advisor: Yu-Tai Ching

Institute of Computer Science and Engineering

National Chiao Tung University

ABSTRACT

In order to understand the fly brain functions, the three-dimensional knowledge of the topography and localization of brain structures and how neuronal circuits to connection is very important. We build a texture-based volume visualization system for confocal microscopic image of *Drosophila's* brain. In order to compare the brain structures between two individuals, our system provides multi-volume rendering. Furthermore the lighting, shadowing effect and pre-integrated volume rendering are supported in the system to make it more sophisticated. Geometric information derived by neuron tracing can be incorporate with our visualization system perfectly.

致　謝

感謝荊宇泰教授這兩年的細心指導，讓我有能力完成這個論文。感謝楊傳凱老師及謝昌煥老師百忙之中抽空來參加我的口試，並給我很好的建議。也感謝昌杰學長帶我體會程式語言和圖學的奧妙。秉璋學長在演算法部分也給我很好的意見及思考方向。

感謝我的家人，念大學和研究所的這段期間很少回家，很想他們。現在終於脫離學生生活了，當兵前的這段時間，當然要留給家人啦。祝我的家人身體永遠健康，也祝我的朋友身體永遠健康

海角七號破了這幾年國片的票房紀錄，在我完成論文口試時，他的票房已經默默的破了四千萬了。國片拍得那麼好看，真是台灣的驕傲。我還去電影院看兩次呢。希望大家要多支持台灣的人事物。台灣很美，台灣好棒！！！

向工呆丸謀人宅！！！

誰說台灣沒人才！！！

Contents

List of Figures

Chapter 1
Introduction

In this thesis, we present a volume data visualization system for confocal microscopic of *Drosophila's* brain build based on using the texture-based volume rendering algorithm. A convenient visualization system which provides sophisticated lighting and shadowing models and convenient observation with multi-volume data can help the process of the bio-medical science. Our visualization system provides these powerful functions with intuitively graphic user interface. These features help biologist to gain detailed and useful insights into the volume data of *Drosophila's* brain.

Texture-based volume rendering [1][2][3][4] is an efficient visualization technique that takes advantages of the texture mapping hardware in the computer video card. In this approach, we process the volume data as a stack of parallel textured slices from back to front. To render the data, the integration of luminance and opacity are left to an image composition step. These steps can be efficiently performed by extensively using of image composition and texturing hardware.

Although the texture-based volume rendering is a commonly used technique in visualization applications, there is a serious drawback caused by non-linear transfer functions. To visualize the region of interesting in the volume data, a non-linear complex transfer function is needed. In this case, we need additional slices for integrating non-linear transfer functions to approximate the volume rendering integral. That implies the lower frame rates would be gained on modern consumer graphics hardware. Klaus Engel [5] introduce a volume rendering integral approach that improves the image quality by pre-integrated volume rendering which allows us to avoid additional slices caused by non-linear transfer functions in a pre-processing step.

Lighting effect gives impressive result in terms of additional realism, and improved spatial comprehension can be achieved. Many applications provide lighting efficiently by an approximation to the Phong local surface shading model [6]. Furthermore, the surface normal information is required when implement Phong model. But no gradient estimation is supported in current consumer hardware when using texture mapping for rendering volumetric data. Allen Van Gelder describes a gradient-based shading criterion [7], in which the gradient magnitude is interpreted directly from volumetric data, and stored in another 3-D texture. In the rendering phase, the gradient 3-D texture could efficiently be combined with texture-based volume rendering and the gradient is used as normal information of lighting model.

Although gradient-based shading criterion provides the normal information of volumetric data, one should store the pre-calculated gradient together with the volumetric data, and then four times memory as the dataset is stored in the graphics hardware. The storage of normal information makes the problem of limitation of the texture memory more badly. Unfortunately, since the surface normal is approximated by the normalized gradient of a scalar field, these methods are unsuitable for shading homogeneous regions. Joe Kniss provide a shadowing mechanism [8][9] for shading homogeneous regions without additional memory space. This technique also significantly improves the visual perception and spatial understanding of volume data.

In order to compare the fly brain structures between two individuals, simultaneously visualization between multi-volume data is needed. We expand texture-based volume rendering technique to provide multi-volume rendering.

In this thesis we present a volume data visualization system for confocal microscopic image of *Drosophila's* brain implemented by texture-based volume rendering. Our system provides basic texture-based volume rendering, high quality pre-integrated volume rendering, and volume rendering with lighting and shadowing. Furthermore, our system can render multi-volume data at the same time. In order to understand the three-dimensional knowledge of how neuronal circuits to connection, geometric information deprived by neuron tracing [10] can be incorporate with our visualization system perfectly.

The structure of this thesis is described as follow. The first chapter gives the motivation and an introduction of our system. In chapter 2, we describe the background material used in our visualization system, including the optical model for direct volume rendering, volume rendering integral, pre-integrated volume rendering, and shading models. In chapter 3, the detail of visualization flowchart and the process of multi-volume rendering are reported. The results of our visualization system are demonstrated in chapter 4. Conclusion and future work are listed in chapter 5.

Chapter 2
Background Materials

In this chapter, we take a brief introduction of the relative research of our visualization system. Section 2.1 reports the optical model for the direct volume rendering algorithm. We assume that any object in space is formed by many small particles like individual molecules. By simulating the state transition when light passing through the volume, we can integrate the change of light intensity to compute the 2-D projective image from 3-D volumetric data. "Volume rendering integral" is a formula derived from above optical model to implement the direct volume rendering algorithm. We then describe the pre-integrated volume rendering algorithm, which improves the artifact of 2-D projective image caused by insufficient sampling rate of the direct volume rendering algorithm.

In section 2.2, we report some shading method; include the lighting model and the shadowing mechanism. Volume rendering with lighting effect gives impressive result in terms of additional realism, but there still some disadvantage with it. The lighting model with gradient as surface normal information is not suitable for homogeneous volumetric data. The shadowing mechanism is a good substitute for the lighting model to improve this problem. We also implement the shadowing mechanism in our visualization system.

2.1. Texture-based Volume Rendering

The using of three-dimensional texture mapping hardware to perform direct volume rendering, so-called "Texture-based Volume Rendering," was described by Cabral. An optical model is build to map the intensity of volumetric data to optical properties, such as color and opacity. During rendering time, optical properties are accumulated along each viewing ray to form a 2-D projective image from the 3-D volumetric data. This algorithm generates images of three-dimensional volumetric data set directly without explicitly extracting geometric surfaces from the original volume data.

In this section, we describe the optical model of direct volume rendering and introduce "Volume rendering integral" [11] which is the formula to compute the accumulated optical properties along viewing ray. We then introduce the implementation of the texture-based volume rendering algorithm on consumer graphics hardware and describe the artifact caused by insufficient sampling rate. Engel presents the pre-integrated volume rendering algorithm to solve this artifact without the performance overhead caused by rendering additional interpolated slices. This algorithm is suited to achieve the goal of high-quality volume rendering at interactive frame-rates on standard PC hardware.

2.1.1 Optical Model and Volume Rendering Integral

Nelson Max proposes an optical model of volume visualization. He assumes that the objects in space are formed by many small particles like individual molecules. Each particle occludes incoming light and adds its own glow defined by its opacity and luminance property. The image of objects to the user's eyes is the result of light ray passing through objects. So the optical properties of particles in the objects influence the light passing through volumetric data. The final projective image is due to the absorption and emission of light from such particles in the data.

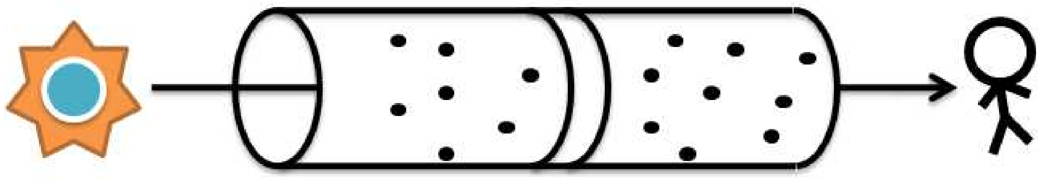
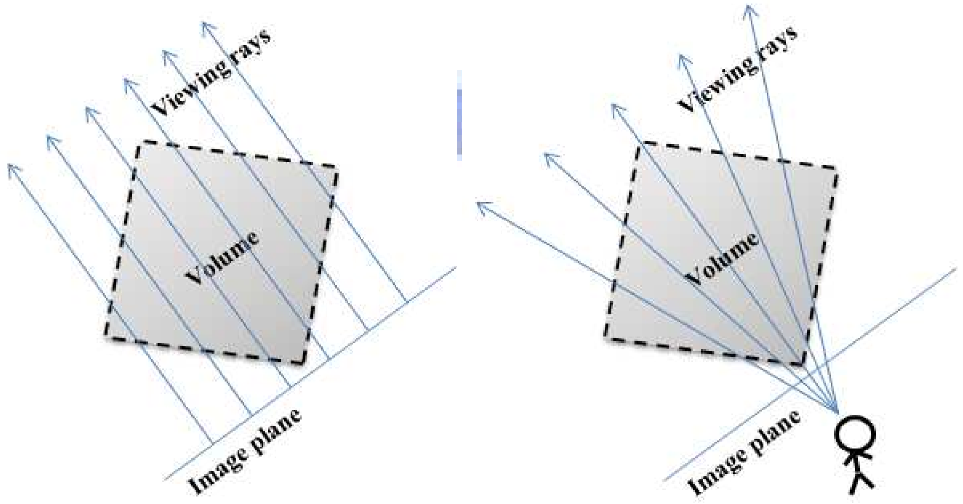


Fig. 2-1: Light ray passes through a volume.

By considering the optical model described above, we can simplify our analysis process of volume lighting computations by taking into account only a single long cylinder centered on the light ray that passes through the volume. As shown in Fig. 2-1. The cylinder is thin enough to assume that the volume properties do not change on its breadth, but they will change on its length. At the back end of this cylinder, background light comes in, and at the front end of this cylinder, light exits and travels to the user's eyes. We can compute such light ray pixel by pixel to generate a 2-D projection image from volumetric data, as shown in Fig. 2-2. The orthogonal projection and perspective projection is suited projection algorithm for this optical model.



(a) Orthogonal projection (b) Perspective projection

Fig. 2-2: Viewing rays passes through a volume

For each viewing ray, the quantity I, which is the amount of light received at one point on the image plane, is:

$$I(D) = \int_0^D c\left(s\big(x(\lambda)\big)\right)\tau\left(s\big(x(\lambda)\big)\right)e^{-\int_0^\lambda \tau\left(s\left(x(\lambda')\right)\right)d\lambda'}d\lambda$$

Here the viewing ray $x(\lambda)$ is parameterized by the distance λ to the eyes. For the visualization of scalar field $s\big(x(\lambda)\big)$, the transfer function is used to defined the color density $c(s)$ and the extinction density $\tau(s)$, which map the scalar value $s(x)$ to color and extinction coefficients. D is the maximum distance, i.e., there is no color density $c\big(x(\lambda)\big)$ for λ greater than D.

We call this formula the volume rendering integral, which is the fundamental element in the direct volume rendering algorithm. For implementation of the volume rendering integral, a numerical integration is required. The most common way to get the approximation of the volume rendering integral is the computation of a Riemann sum for n equal ray segments of length $d = {}^{D}\!/_{n}$. The approximate evaluation of the volume rendering integral can be stated as:

$$I = \sum_{i=0}^{n} \alpha_i c_i \prod_{j=0}^{i-1}(1-\alpha_j)$$

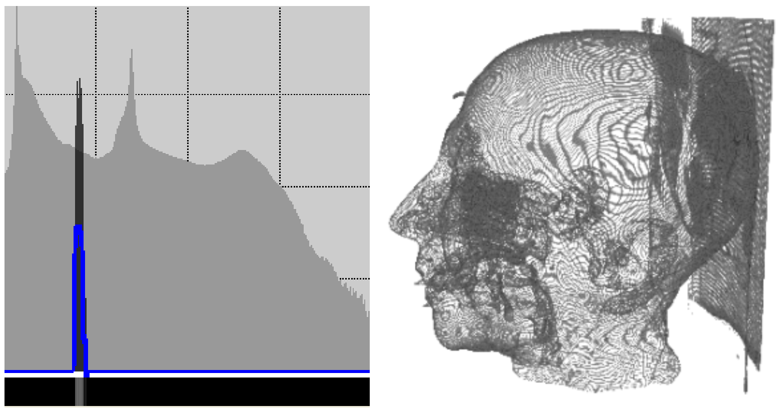
In the approximate evaluation of the volume rendering integral, opacity α_i approximates the absorption, and color c_i approximates the emission at sample i. The product in the sum represents the amount of light attenuated at sample i before reaching the user's eyes and the sum of the volume rendering integral accumulate the light effect when light passing through the volume.

2.1.2 Pre-integrated volume rendering

Volume visualization is implemented by integrating the color and opacity values across the 3D volume data. This integration in the texture-based volume rendering algorithm is performed by sampling the volume with parallel textured polygons, called proxy geometry, at regular intervals.

According to the sampling theorem, a correct reconstruction is only possible with the sampling rates larger than the Nyquist frequency. In the volume visualization algorithm, it is sufficient to sample at the resolution of the scalar field to avoid aliasing with respect to scalar value. However, the scalar field is sampled before being transformed by a transfer function. Non-linear transfer functions may add arbitrary frequencies to the data and increase the sampling rate required for the volume rendering integral. The higher sampling rates of the volume visualization algorithm are requires for capturing all details.

Consider a thin spike in the transfer function, this spike results a very thin surface by the volume rendering algorithm. If the feature defined by transfer function is smaller than the sampling range, which is often encountered when drawing an iso-surface, some rays will sample the detail and others will miss it completely. The result of volume visualization is a series of aliasing bands rather than a continuous surface. If the feature is somewhat larger than the sampling range, a similar problem is still occurred because some rays will sample this feature once while others will sample it twice, as shown in Fig. 2-3. Fig.2-3a is a transfer function with a very thin spike, and Fig.2-3b displays the result of visualization.



(a) (b)

Fig. 2-3: Sampling rate is smaller than the Nyquist frequency

Such artifacts can be reduced in traditional volume renderings by sampling at high rates or using smooth, low-frequency transfer functions such as Gaussian curve to blur the feature. Unfortunately, high sampling rates induces the computation overhead and limit the performance of hardware and software. And smooth transfer functions limit the types of renderings result. Both of such improvements still not guarantee a sufficient sampling.

The pre-integrated transfer functions algorithm solve this problem by pre-computing a 2-D table that stores the integral result of all possible sampling pairs of volume rendering. This table is then indexed during rendering by each ray sampling pair received from neighbor sampling slice, as shown in Fig. 2-4. The pre-integration volume rendering algorithm assumes that the transfer function between any two discrete sampling pairs is linear. Looking up the pre-integrated lookup table for any two sampling pairs guarantees that no transfer function detail is ignored.

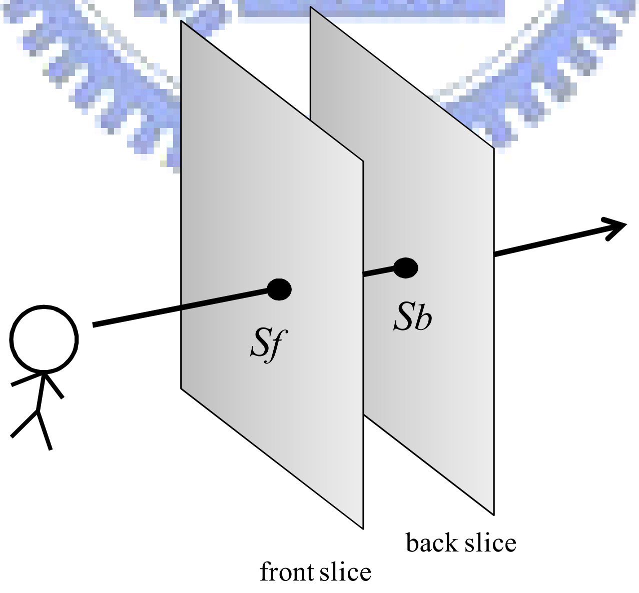


Fig. 2-4: Front and back slice of the pre-integrated volume rendering

2.2. Volume rendering with Shading

In this section we introduce some shading technique, include lighting and shadowing. Lighting is a basic shading method and is easy to implement, but the additional memory space is needed to store the normal information. Further, the normal gained by gradient is undefined in homogeneous regions. Shadowing provides high quality rendering result without extra memory space. It also provides currently feasible solution for homogeneous volume data.

2.2.1 Lighting

For improving the quality of volume rendering, sophisticated shading model is required to capture characteristics of volume data and provide subtle lighting effects. A local illumination models can approximate the light intensity on the surface of an object by considering the lightings effects in three different ways, emission, transmission, and reflection. This model is evaluated as a function of the normal of the surface with respect to the position of a point light source and some material properties. Indirect light and shadows are not taken into account. The most popular lighting model is Phong model:

$$I = k_a l_a + k_d l_d(\vec{l} \cdot \vec{n}) + k_s l_s(\vec{h} \cdot \vec{n})^n$$

Which computes the reflected intensity as a function of local surface normal $\vec{n}$, the lighting direction $\vec{l}$, ambient, diffuse, and specular intensity l_a, l_d, l_s of the light source, ambient, diffuse, specular, and shininess coefficients k_a, k_d, k_s, n of the object, and the half-vector $\vec{h}$ of lighting and viewing direction.

The gradient information is usually used as normal during rendering. The central differences at each voxel are used to gain the normal vector. The method of central differences approximates the gradient as the difference of data values of two voxel neighbors along a coordinate axis, divided by the physical distance h. The following formula computes the x, y, and z component of the gradient vector at voxel location $\vec{P}_{(i,j,k)}$, individually. $v(\vec{P})$ is the function to get the value of volume data.

$$g_x(\vec{P}_{(i,j,k)}) = \frac{v(\vec{P}_{(i+1,j,k)}) - v(\vec{P}_{(i-1,j,k)})}{2h}$$

$$g_y(\vec{P}_{(i,j,k)}) = \frac{v(\vec{P}_{(i,j+1,k)}) - v(\vec{P}_{(i,j-1,k)})}{2h}$$

$$g_z(\vec{P}_{(i,j,k)}) = \frac{v(\vec{P}_{(i,j,k+1)}) - v(\vec{P}_{(i,j,k-1)})}{2h}$$

The normal information computed from volume data is stored in a 3-D array. It is efficient to combine the normalized gradient and original volumetric data into a single RGBA texture to reduce the cost of texture lookup and interpolation.

2.2.2 Shadowing

Furthermore, the normal required for the Phong model is derived from the normalized gradient of the scalar field. For many volumes, homogeneous regions pose problems for typical gradient based surface shading. While this normal is well defined for the regions in the volume that have high gradient magnitudes, this normal is undefined in the homogeneous regions, where the gradient may be the zero vector. The use of the normalized gradient is also troublesome in the regions with low gradient magnitudes, where noise can significantly degrade the gradient computation.

Kniss provides a shadowing technique with two important characteristics. First, the slice axis of proxy geometry is modified from the viewing direction to the direction halfway between the lighting and viewing directions. This allows the same slice to be rendered from both the eye and light points of view. Second, an off screen rendering buffer, called light-buffer, is needed to accumulate the amount of light attenuated from the light's point of view.

Chapter 3
Overview of Our Visualization System

In this chapter, we introduce each part of our visualization system. Our system can be used to interactively visualize multi-volume data with pre-integrated transfer function, lighting effect, and shadowing effect. The geometry data produced by neuron tracing can also be combined with our volume visualization system perfectly.

3.1. Volume visualization

3.1.1 Texture-based volume rendering

The texture-based volume rendering algorithm is a favorite technique to implement the volume rendering integral. It is the technique we used to efficiently visualize volumetric data by using texture mapping hardware. The flowchart of the texture-based volume rendering algorithm is shown in Fig. 3-1.

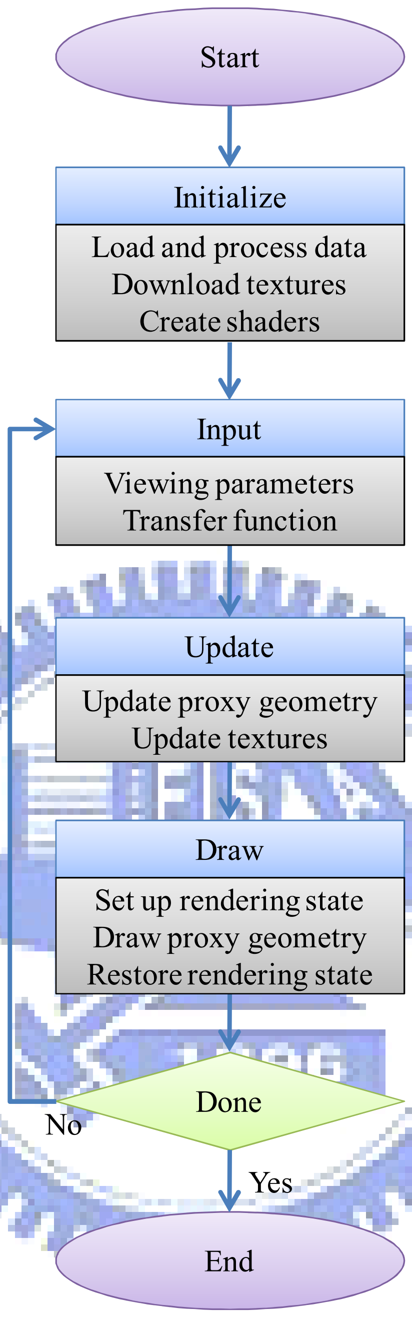


Fig. 3-1: Flowchart of the texture-based volume rendering algorithm

At the beginning of our system, volume data are loaded and stored in the CPU memory as a single 3-D array. Then they are padding to power-of-two-size texture to maximize rendering performance and downloaded to graphics memory [12]. Transfer function texture and fragment shaders [13] are also created in the initialize stage. Notice that this stage is usually performed only once in our system.

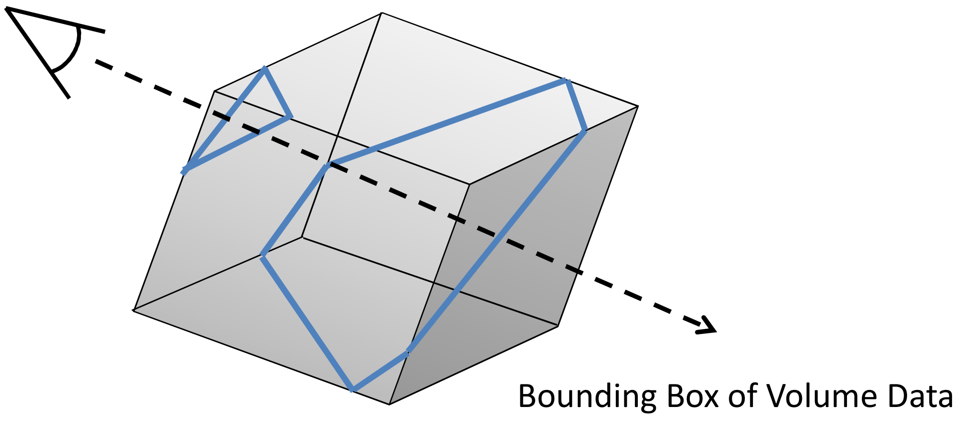


Fig. 3-2: Proxy geometry of texture-based volume rendering

After our system receives user inputs, the proxy geometry is computed and stored in a proxy pool in the update stage. Corresponding to the texture-based volume rendering algorithm, the image of the volume data are created by drawing and compositing the proxy geometry in sorted order. The proxy geometry is parallel textured polygons and they are gained by firstly calculating the intersections between each parallel plane which is vertical to the viewing direction and the edges of the volume bounding box. Then the intersected vertices of each plane are sorted in a counterclockwise direction around their center. The resulting is a set of polygons for sampling the volume data. Fig. 3-2 illustrates the calculating process with two slice polygons. The first polygon contains three vertices and the second is composed of six vertices. For each vertex, the corresponding 3D texture coordinate is also calculated. The calculations of proxy geometry are all done by the CPU.

During the update stage, the textures of transfer function lookup table are refreshed if the transfer function is changed. The transfer functions are used to emphasize the futures of the data by mapping the value of data to optical properties. Typically, these transfer functions are implemented with 1-D texture lookup tables. When the lookup table is built, color and opacity are usually assigned separately by the transfer function.

Before the drawing stage, the alpha blending operator needs to be set up to accumulated color and opacity. If the slice polygons are rendered in back-to-front order, a single step of the compositing process in back-to-front order is known as the "Over" operator [14]:

$$c_{final} = c_{source} + (1 - \alpha_{source}) \cdot c_{destination}$$

$$\alpha_{final} = \alpha_{source} + (1 - \alpha_{source}) \cdot \alpha_{destination}$$

Where $c_{destination}$ and $\alpha_{destination}$ are the color and opacity in the frame-buffer. c_{source} and α_{source} are the color and opacity obtained from the fragment shading stage. c_{final} and α_{final} are the accumulated color and opacity. The final image is computed along the viewing ray from the back of the volume.

If slice polygons are sorted in front-to-back order, the "Under" operator is used:

$$c_{final} = (1 - \alpha_{destination}) \cdot c_{source} + c_{destination}$$

$$\alpha_{final} = (1 - \alpha_{destination}) \cdot \alpha_{source} + \alpha_{destination}$$

Where $c_{destination}$ and $\alpha_{destination}$ are the color and opacity in the frame-buffer. c_{source} and α_{source} are the color and opacity obtained from the fragment shading stage. c_{final} and α_{final} are the accumulated color and opacity from the front of the volume.

```
uniform sampler3D uTexVoxel;
uniform sampler1D uTexTf1d;

void main(){
  float Voxel = texture3D(uTexVoxel, gl_TexCoord[0].xyz).r;
  gl_FragColor = texture1D(uTexTf1d, Voxel).rgba;
}
```

Fig. 3-3: The shader code of the texture-based volume rendering algorithm

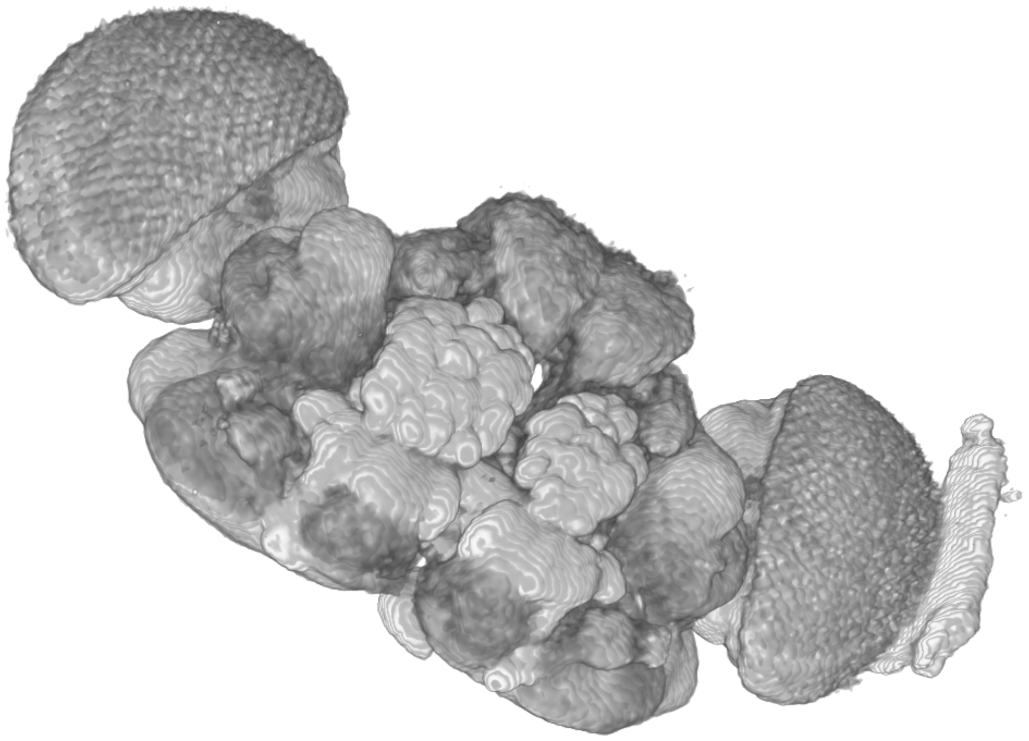


Fig. 3-4: The result of basic texture-based volume rendering

In the drawing stage, the slice polygons are rasterized and blended into the frame buffer in sorted order. In the fragment shading stage, the interpolated 3-D texture coordinate of each fragment is used for looking up the texture of volume data. Then the data value gained by sampling the volume data is used as 1-D texture coordinates for looking up the texture of transfer function. The shader code of the texture-based volume rendering algorithm is shown in Fig. 3-3. After each slice polygon is rendered, it is sent to the compositing stage of the rendering pipeline. Each slice polygon is rendered once in back-to-front or front-to-back order with corresponding blending function in the frame-buffer. Then the projective image of volume data is gained, as shown in Fig. 3-4.

3.1.2 Pre-integrated volume rendering

For contacting the artifact caused by low sampling frequency, we use the pre-integrated lookup table to replace 1-D transfer function lookup table. As the transfer function changed, the corresponding pre-integrated lookup table is calculated and stored as a 2-D texture. In the fragment shading stage, the interpolated 3-D texture coordinate of each fragment is used as the sampling point in the front slice. The sampling point in the back slice is calculated by front slice, viewing direction, and slice interval. These two scalar values are used as a 2-D texture coordinate for a third texture fetch operation, which performs the lookup of pre-integrated colors and opacities from the 2-D texture map of the pre-integrated lookup table. The relationship of these sampling points is shown in Fig. 3-5. The scalar value on the front (back) slice for a particular viewing ray is called s_f (s_b). "uBackSliceDir" is the vector from front to back slice.

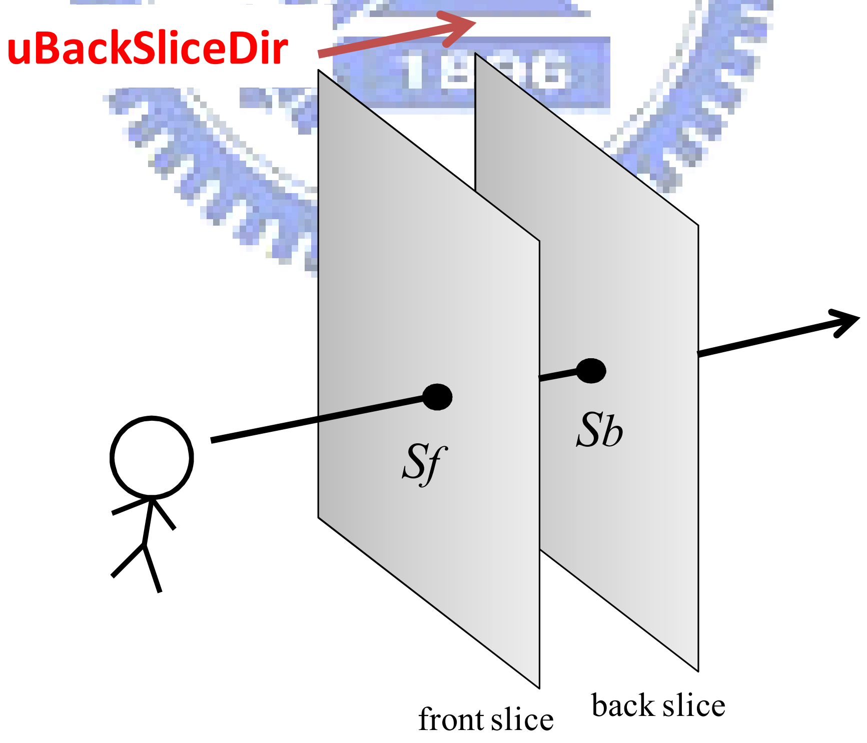


Fig. 3-5: A slab of the volume data between two slices.

The corresponding GLSL code of the pre-integrated volume rendering algorithm is shown in Figure 3-6. The texture coordinate of each fragment on the front slice is gained by interpolated 3-D texture coordinate. The texture coordinate of each fragment on the back slice is gained by front coordinate and uBackSliceDir. The front and back texture coordinates are used as 2-D coordinate (TexPreItgCoord) to fetch pre-integrated lookup table (uTexTfPreItg). The result of the pre-integrated volume rendering is shown in Fig. 3-7.

```
uniform  sampler3D  uTexVoxel; // volume data
uniform  sampler2D  uTexTfPreItg; // pre-integrated lookup table
uniform  vec3  uBackSliceDir; // vector from front to back slice

void main(){
  vec3  TexVoxelCoordFront  =  gl_TexCoord[0].xyz; // interpolated 3-D texture coordinate
  float  VoxelFront  =  texture3D(uTexVoxel,  TexVoxelCoordFront ).r;

  vec3  TexVoxelCoordBack  =  TexVoxelCoordF  -  uBackSliceDir;
  float  VoxelBack  =  texture3D(uTexVoxel, TexVoxelCoordBack ).r;

  vec2  TexPreItgCoord;
  TexPreItgCoord.x  =  VoxelFront ;
  TexPreItgCoord.y  =  VoxelBack

  gl_FragColor  =  texture2D(uTexTfPreItg,  TexPreItgCoord).rgba;
}
```

Fig. 3-6: Shading code of pre-integrated volume rendering

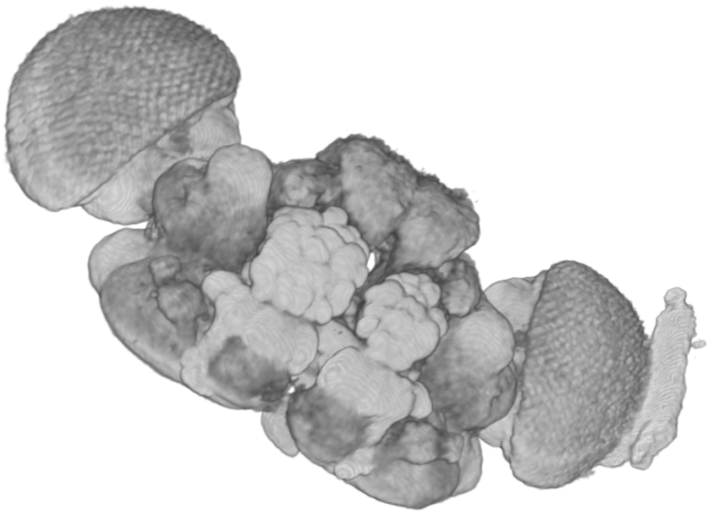


Fig. 3-7: The result of pre-integrated volume rendering algorithm

3.1.3 Volume rendering with lighting

After loading volume data into our visualization system, the surface normal vector of each voxel is gained by computing the central difference at it. Then the surface normal information is stored in a 3-D array. This 3-D array can be incorporated with the original volume data to a single 3-D RGBA texture, and downloaded to graphics memory. Notice that the value of normal vector should be normalized from 0 to 1 to achieve the goal of maximizing rendering performance. The step of computing and downloading the normal information to graphics memory is performed in initialize stage only once in our system.

In the fragment shading stage, the textures of volume, normal, and transfer function lookup table, and coefficients of lighting model are loaded into shaders. Illumination techniques may modify the resulting color before it is sent to the compositing stage of the pipeline. We present the most common shading model, Phong model, which computes the reflected intensity as a function of local surface normal, lighting direction, and coefficient of light. The corresponding GLSL code is shown in Fig. 3-8. Notice that the value of normal vector is shift from -0.5 to 0.5 in the shader to restore the original normal direction. The result of volume rendering with lighting is shown in Fig. 3-9.

```
uniform sampler3D uTexVoxelNormal;
//light coefficients;
//...

void main(){
   vec3 TexVoxelNormalCoord = gl_TexCoord[0].xyz;
   // channel r of uTexVoxelNormal stores the volume data
   // channel gba stores the normal information
   vec4 VoxelNormal = texture3D(uTexVoxelNormal, TexVoxelNormalCoord ).rgba;
   vec4 Fragment;
   // compute Fragment by transfer function
   //...

   vec3 Normal;
   // shift the value of normal from 0 to 1 to -0.5 to 0.5
   Normal.x = VoxelNormal.g - 0.5;
   Normal.y = VoxelNormal.b - 0.5;
   Normal.z = VoxelNormal.a - 0.5;
   // perform Phong model
   //...

   gl_FragColor = Fragment;
}
```

Fig. 3-8: GLSL code of texture-based volume rendering with lighting

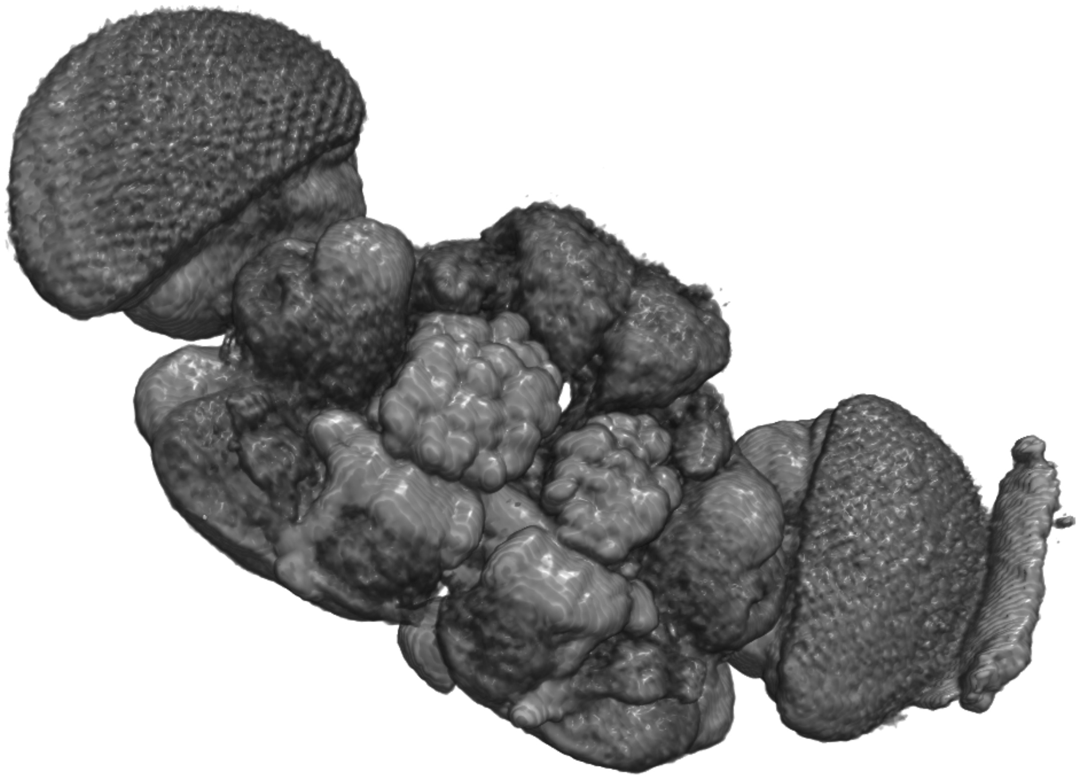


Fig. 3-9: The result of texture-based volume rendering with lighting effect

3.1.4 Volume rendering with shadowing

For the confocal microscopy image of *Drosophila's* brain, homogeneous regions pose problems on the typical gradient based surface shading. The surface normal vectors are well defined in the regions with high gradient magnitudes. For homogeneous regions, the gradient vector may be the zero vectors. The using of the normalized gradient vectors is also troublesome in regions with low gradient magnitudes, where noise can significantly degrade the normalization of gradient vectors.

We implement the shadowing algorithm provided by Kniss. This algorithm has two important characteristics. First, the slice axis of proxy geometry is modified from viewing direction to the direction halfway between the viewing and lighting directions, as shown in Fig. 3-10. When the dot product of the lighting and viewing directions is positive, we slice volume data along the vector halfway between the lighting and viewing directions, seen in Fig. 3-10a. In this case, the proxy geometry of the volume data is rendered in front to back order with respect to the observer. When the dot product is negative, we slice along the vector halfway between the lighting and the inverted viewing directions, seen in Fig. 3-10b. In this case, the proxy geometry is rendered in back to front order with respect to the observer. In both cases, the proxy geometry is rendered in front to back order with respect to the light.

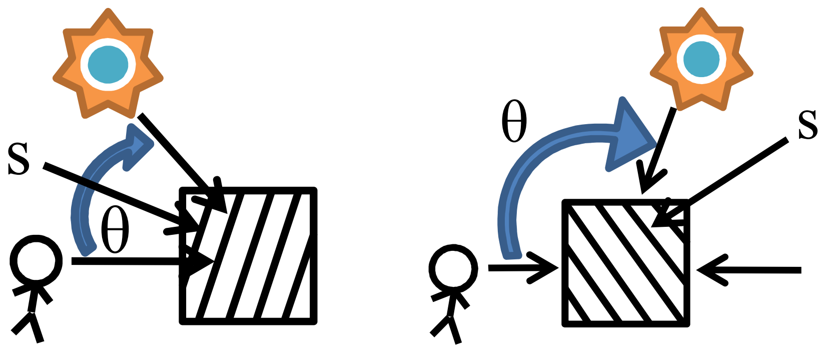


Fig. 3-10: Slice axis dependent on view and light directions

Second, an off screen render buffer, light-buffer, is utilized to accumulate the amount of light attenuated from the light's point of view. This buffer is initialized to light intensity. It can also be initialized using an arbitrary image to create effects such as spotlights.

For the implementation of light-buffer with hardware, we introduce a powerful technique, the frame-buffer object (FBO) [15]. FBO is an extension of OpenGL for doing flexible off-screen rendering, include rendering to a texture. It allows result of rendering to a frame-buffer to be directly read as a texture. FBO takes advantage of good performance because the step of copy from a frame-buffer to a texture is avoided.

Our shading process is shown in Fig. 3-11. At the beginning of rendering, the sampling direction and sorting order of slices are decided. The slice axis is modified for the requirement of rendering the same proxy geometry from both the eye and light points of view.

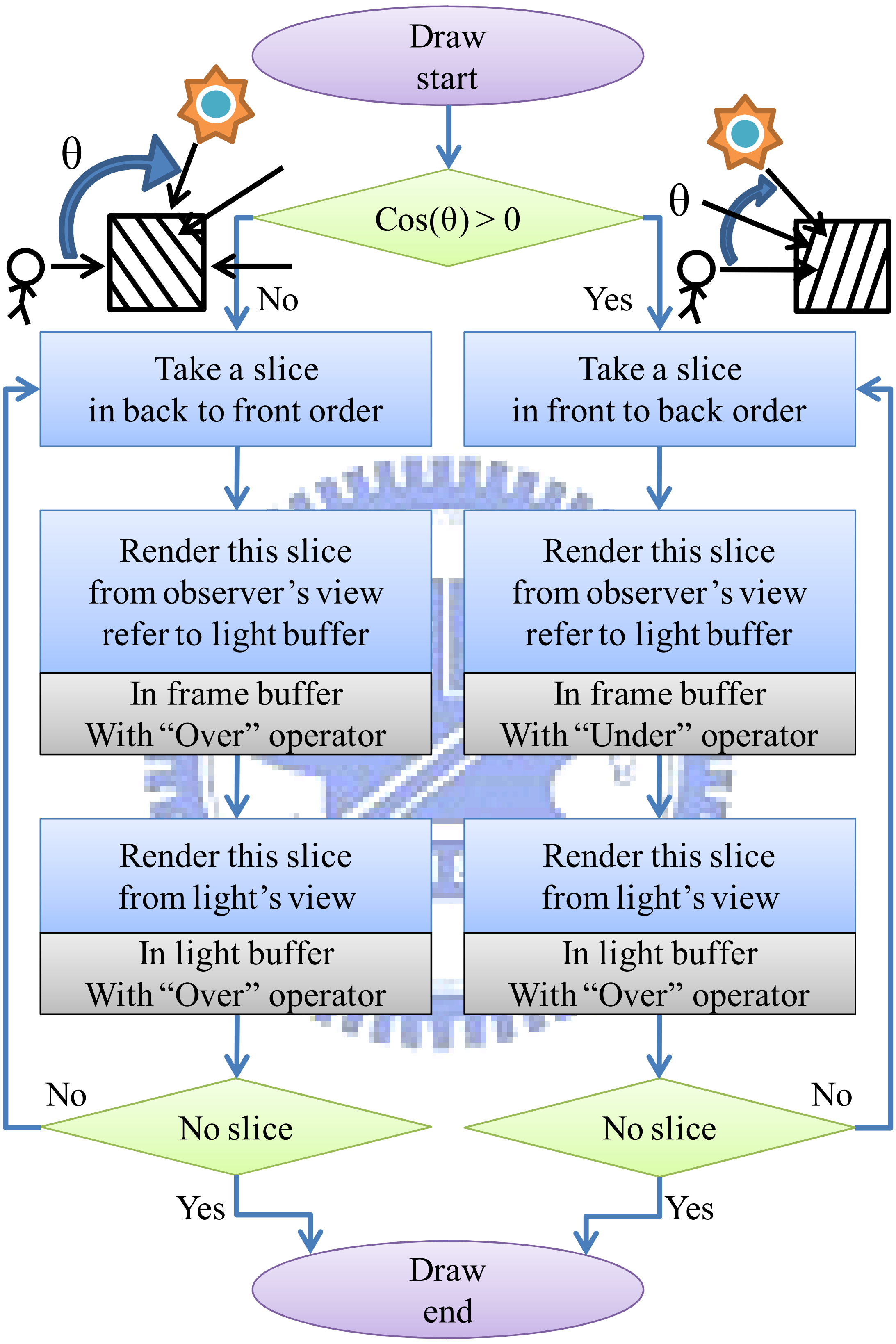
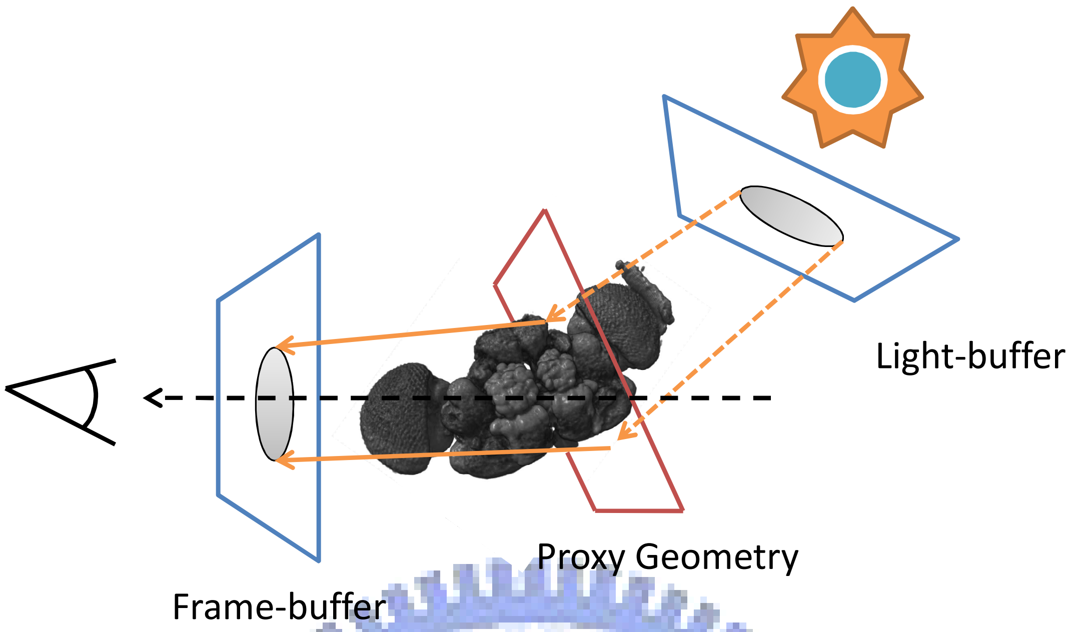
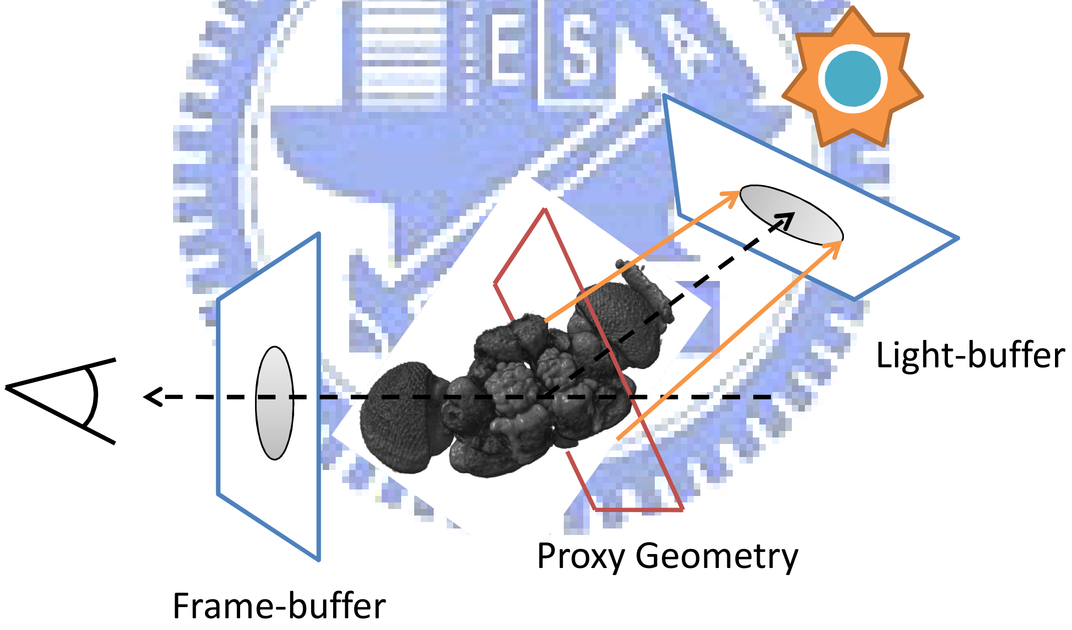

Draw start
θ
Cos(θ) > 0
θ
No
Yes
Take a slice in back to front order
Take a slice in front to back order
Render this slice from observer's view refer to light buffer
In frame buffer With "Over" operator
Render this slice from observer's view refer to light buffer
In frame buffer With "Under" operator
Render this slice from light's view
In light buffer With "Over" operator
Render this slice from light's view
In light buffer With "Over" operator
No
No slice
No slice
No
Yes
Yes
Draw end

Fig. 3-11: Shadowing process



(a)



(b)

Fig. 3-12: Two passes of shadowing algorithm

In the shading stage, each proxy polygon is rendered twice. In the first pass, proxy polygon is firstly rendered from the observer's point of view in the frame-buffer. The light intensity at each fragment of this polygon is acquired by sampling the position it is projected in the light-buffer. The light intensity is used to modulate the brightness of the fragment. In this pass, polygons are blended with over operator if the dot product of lighting and viewing vector is negative, or with under operator if the dot product is positive. This pass is illustrated in Fig. 3-12a.

In the second pass, this polygon is rendered from the light's point of view in the light-buffer to achieve the goal of accumulating the intensity of the light arriving in the first pass of the next polygon. Each fragment of this polygon is firstly sampling the texture of the volume data to get the interpolated data value. Then the data value is used as texture coordinate to sample the texture of the transfer function. Only the opacity of this fragment lookup from the transfer function is required. The fragment is rendered with black color and the corresponding opacity with over operator to achieve the goal of attenuating the intensity of the light. This pass is illustrated in Fig. 3-12b. An example of volume rendering with shadowing can be seen in Fig. 3-13.

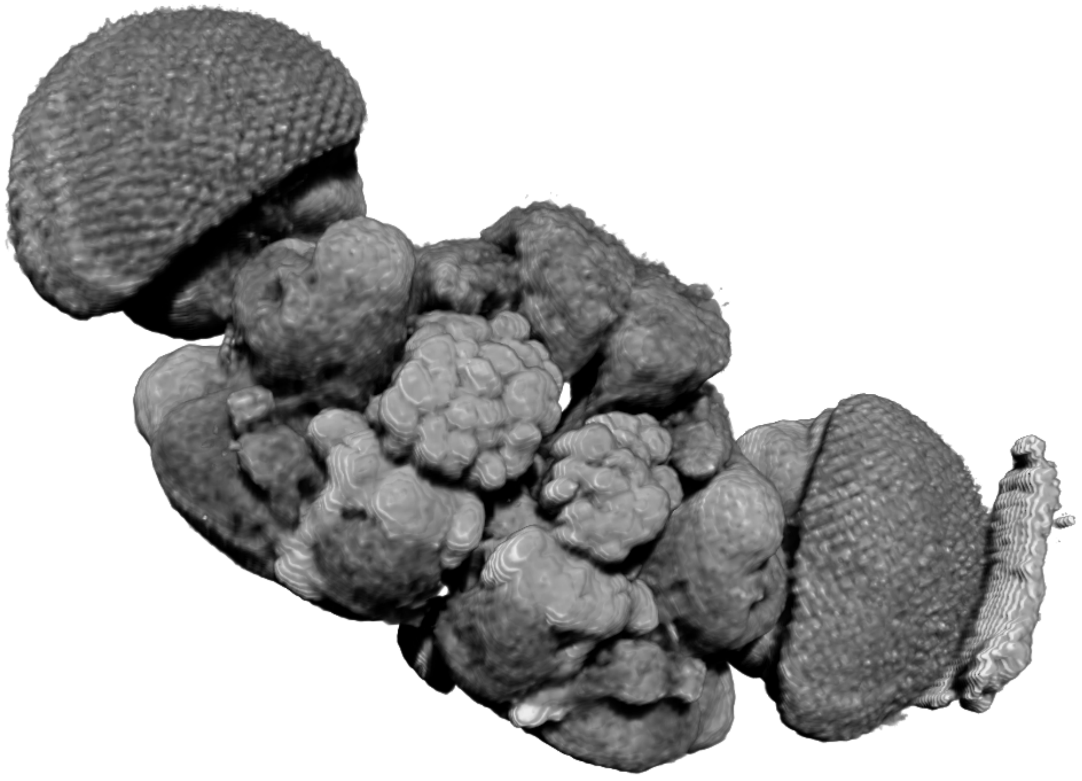


Fig. 3-13: Texture-based volume rendering with shadowing effect

3.1.5 Volume rendering with geometry information

In the neuron study of *Drosophila's* brain, the morphology and geometry of neuron play important roles. Mostly the neuron is with complicated structure and the image volume is derived from confocal microscope slice by slice and this makes it hard to visually observe the neuron. In order to observe the neuron structure we combine the traced neuron branch curve data and volume data to help the user to observe the neuron.

In traditional way, the geometry information is stored as vertices and edges. If we render such geometry information as segments, the depth and space relation is not easy to understand. Display the geometry information as pipe is better than segment because pipe can provide normal information and light shading is possible.

For rendering segments as pipes, each segment is shown as cylinder constructed by polygons, and each end of segment is shown as spheres. We use an adjustable radius to calculate the corresponding cylinders and spheres, as shown in Fig. 3-14.

To achieve the goal of blending the geometry information into the volume rendering, the polygons of geometry information needs to be drawn before the volume data with "Over" operator. The depth test of rendering pipeline culls the fragments of volume rendering that are behind geometry data. If the "Under" operator is used, render the geometry data and the volume data into separate frame-buffers and composite two frame-buffers at the end. In this case, the depth values from the geometry data are used for culling fragments in the volume rendering. The result of volume rendering with geometry information is shown in Fig. 3-15.

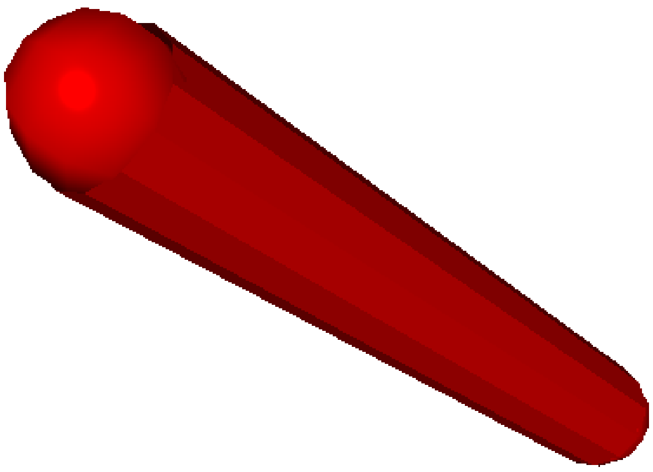


Fig. 3-14: Display segment as pipe

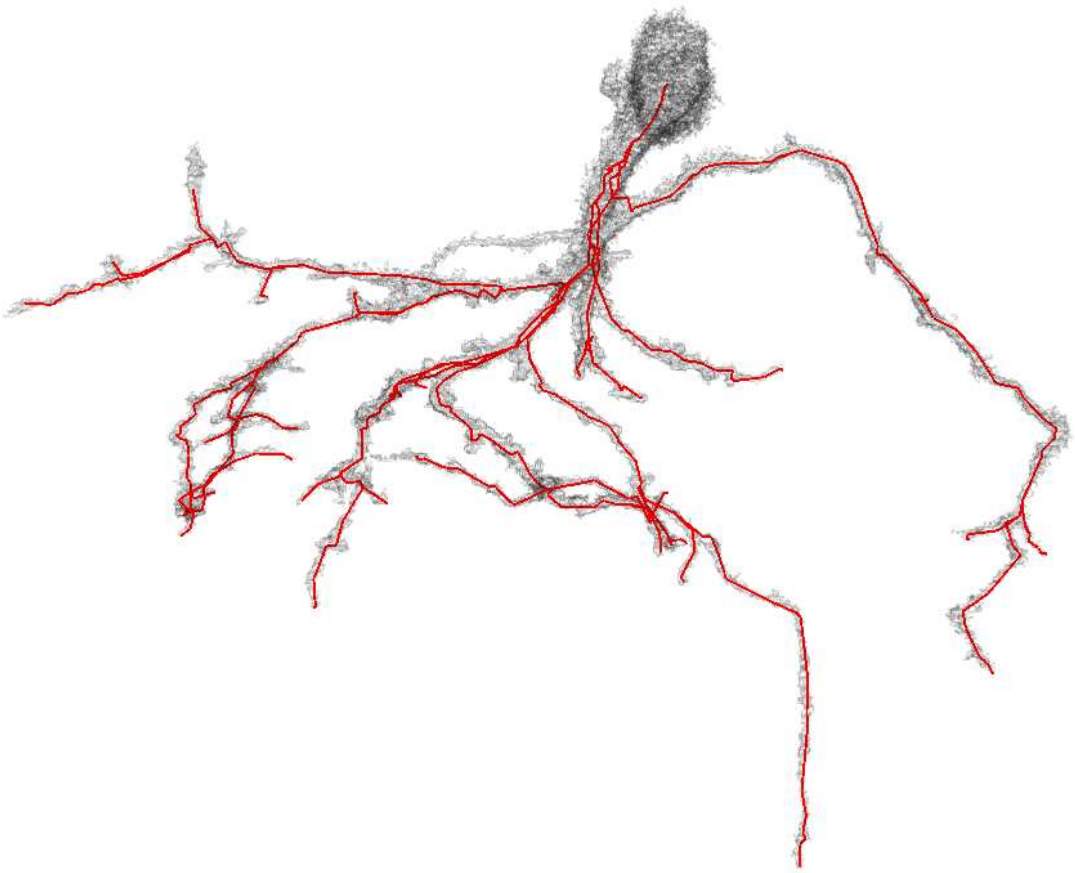


Fig. 3-15: A neuron *Tadpole*

3.2. Multi-volume rendering

In order to compare the brain structures between two individuals, it is in general essential to show all the volume data concurrently. For example, compare the different neuron locations in a fly brain between different experiments to understand how neuronal circuits to connection. Multi-volume rendering provide better observation for bio-medical science.

For rendering multi-volume data at the same time, the visualization stages of our system have a little modification from the process shown before. In the initialize stage, all volume data are loaded, processed and pushed into a volume pool. In the update stage, each volume data in the volume pool is sliced along the sampling direction, and all of such slices are pushed into a proxy pool. We sort all slices in the proxy pool; no matter what volume data they are corresponded.

Our visualization system allows user to selectively set individual rendering coefficients such as transfer functions and material of lighting for different volume data. The deform matrix of each volume data can be set individually for providing better eyeshot and observation. User can also decide the usage of pre-integrated lookup table and lighting effect. See Fig. 3-16 for an example, where several different rendering coefficients have been set to visualize multi-volume data of a human head and a fly brain.

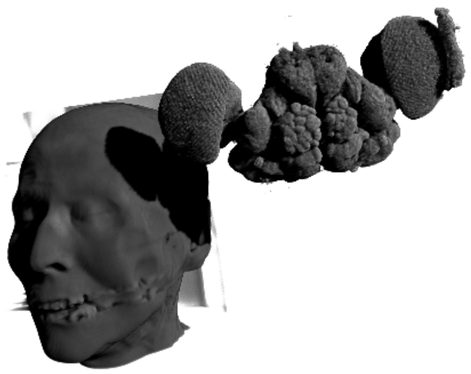


Fig. 3-16: The result of multi-volume rendering

3.3. Summary

In chapter 3.3, we introduce the flowchart of our visualization system; include the flowchart of data processing and the shading effects.

3.3.1 Flowchart of Our Visualization System

Fig. 3-17 shows a flowchart diagram illustrating the complete procedure of our visualization system for the data processing and visualization process. In the initialize stage, volumetric data and geometric information are loaded. In this stage, the texture and of each volumetric data are created and the polygons of geometry information are calculated. After user input the parameters of lighting, viewing, and rendering, and define the transfer function, we create the proxy geometries by sampling each volume data along the sampling direction in the update stage. Then we render all proxy geometry with the shading coefficient defined by user in the render stage. Thus, we complete one pass of rendering.

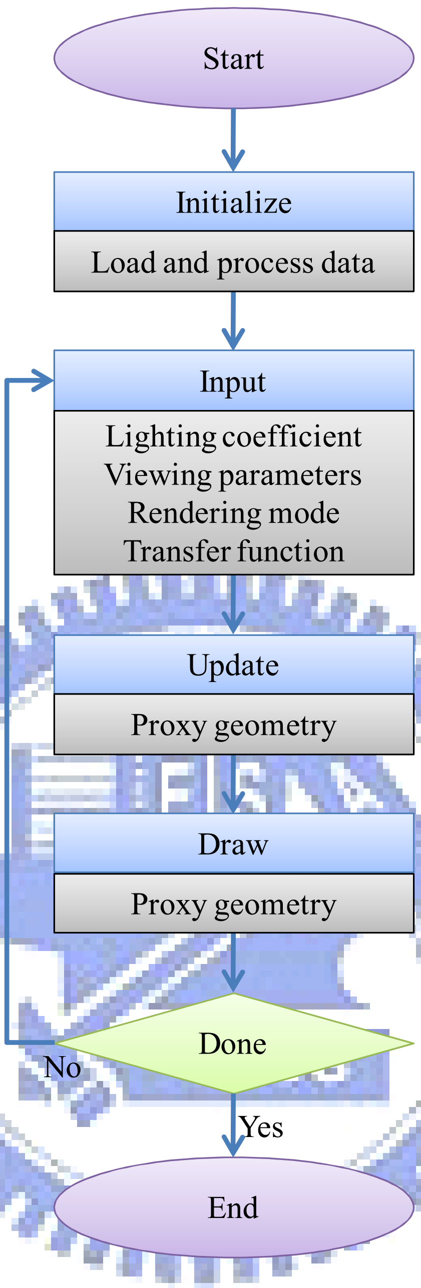


Fig. 3-17: Main process of our visualization system

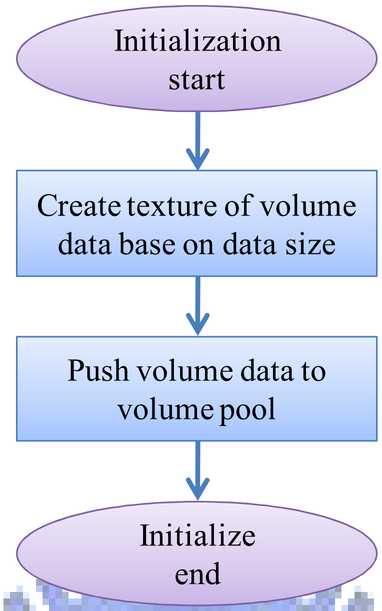

Initialization start
Create texture of volume data base on data size
Push volume data to volume pool
Initialize end

Fig. 3-18: Initialization stage of main process

3.3.2 Initialization Stage

In the initialization stage, as shown in Fig. 3-18, volume data and the corresponding geometry data are loaded. If the lighting effect is enabled, the normal information of volume data is calculated in this stage. For multi-volume rendering, we build a volume pool and push all volume data into it.

3.3.3 Update Stage

In this stage, the proxy geometry of each volume data is calculated. First, the sampling direction is decided. Then every volume data is sampled from the volume pool along sampling direction. All proxy geometry gained in this stage are pushed into a proxy pool and rendered in the next stage.

Before deciding the sampling direction, the shading coefficient of the shadowing effect must be checked firstly. If the shadowing effect is enabled, the sampling direction is decided by the locations of light and observer. When the angle between lighting direction and viewing direction is smaller than 90°, it means that the dot product of the lighting and viewing directions is positive, the sampling direction is the vector halfway between the lighting and viewing directions. When the angle between lighting and viewing direction is larger than 90°, it means that the dot product of the lighting and viewing direction is negative, the sampling direction is the vector halfway between the lighting and inverted viewing directions. On the other way, if the shadowing effect is disabled, the sampling direction is the viewing direction.

After deciding the sampling direction, pick one volume data from volume pool. The volume data is deformed by it's deform matrix, which is decided by the translation, rotation, and scale of this volume. After deforming, the proxy geometry of this volume data is gained by sampling along the sampling direction. All volume data in the volume pool are processed by the procedure as described before.

As all volume data in volume pool have been processed, the proxy geometry in the proxy pool should be sorted. The rule of sorting is decided by the distance between proxy geometry and observer. The procedure of this stage is shown in Fig. 3-19.

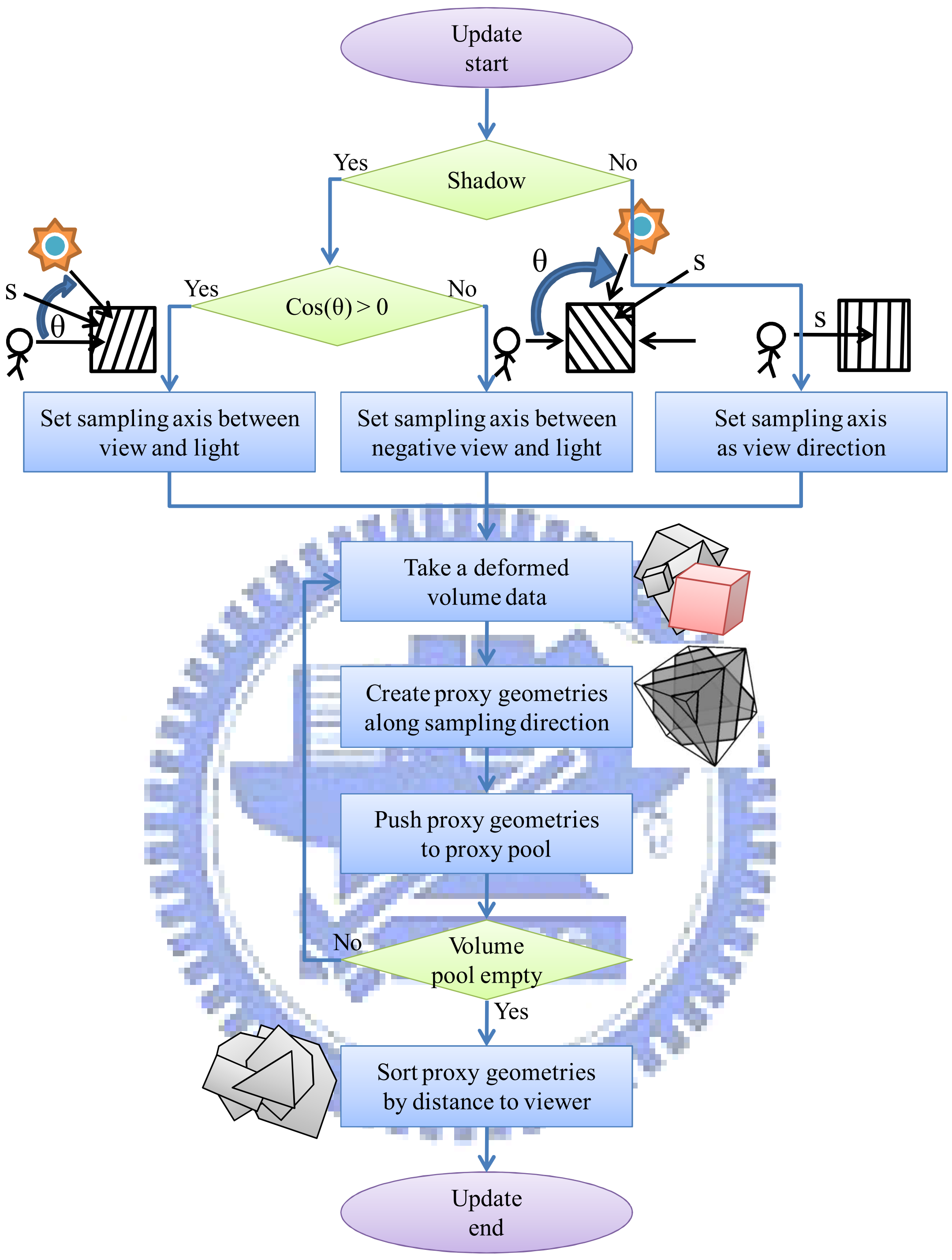

Update
start
Shadow
Yes
No
Cos(θ) > 0
Yes
No
s
θ
θ
s
s
Set sampling axis between view and light
Set sampling axis between negative view and light
Set sampling axis as view direction
Take a deformed volume data
Create proxy geometries along sampling direction
Push proxy geometries to proxy pool
Volume pool empty
No
Yes
Sort proxy geometries by distance to viewer
Update
end

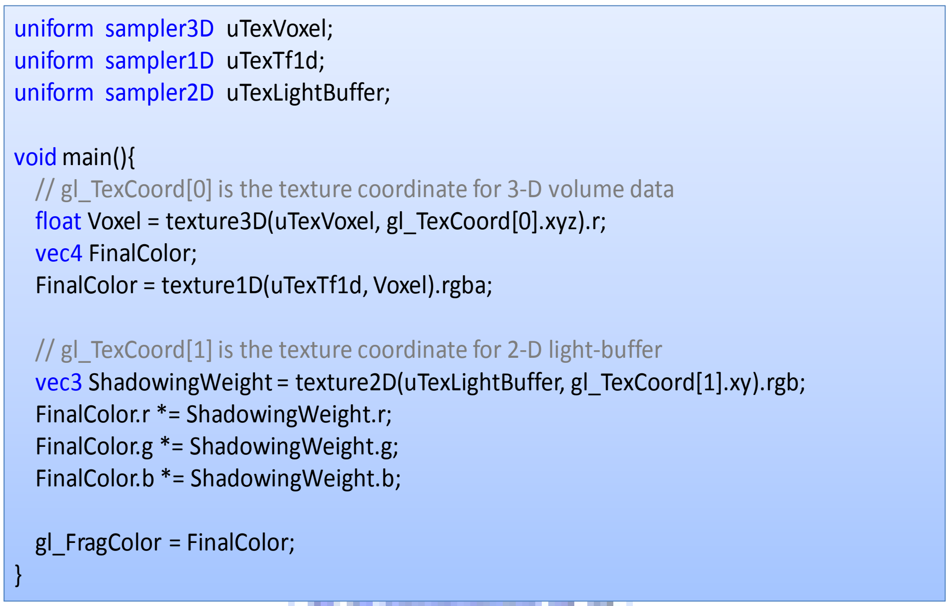
Fig. 3-19: Update stage of main process

3.3.4 Draw Stage

In this stage, the result of the volume visualization is created. First, check if the shadowing effect is enabled. If not, the proxy geometry in the proxy pool is rendered in back to front order (by the distance between proxy geometry and observer from near to far) in the frame-buffer with Over operator.

If the shadowing effect is enabled, the locations of light and viewer should be checked. If the dot product of the lighting and viewing direction is positive, we start the two-pass rendering procedure as follow. Pick proxy geometry in the proxy pool in front to back order. First, render it in the frame-buffer with Under operator. In this pass, the corresponding coefficient of shadowing of each fragment is lookup in the light-buffer. And the final color of each fragment is modified by this coefficient. In the second pass, render the same proxy geometry in the light-buffer with Over operator to accumulate and calculate the shadow coefficient. This pass can be done efficiently with the technique called frame-buffer object. After the two-pass rendering procedure, if the proxy pool is not empty, repeat this procedure.

If the dot product of the lighting and viewing direction is negative, the proxy geometry in the proxy pool is picked in back to front order. In pass one; each proxy geometry is firstly rendered in the frame-buffer with Over operator. In pass two, the same proxy geometry is rendered in the light-buffer with Over operator, two. The corresponding shader code is shown in Fig. 3-20. The flowchart of this stage is shown in Fig. 3-21.



```
uniform sampler3D uTexVoxel;
uniform sampler1D uTexTf1d;
uniform sampler2D uTexLightBuffer;

void main(){
   // gl_TexCoord[0] is the texture coordinate for 3-D volume data
   float Voxel = texture3D(uTexVoxel, gl_TexCoord[0].xyz).r;
   vec4 FinalColor;
   FinalColor = texture1D(uTexTf1d, Voxel).rgba;

   // gl_TexCoord[1] is the texture coordinate for 2-D light-buffer
   vec3 ShadowingWeight = texture2D(uTexLightBuffer, gl_TexCoord[1].xy).rgb;
   FinalColor.r *= ShadowingWeight.r;
   FinalColor.g *= ShadowingWeight.g;
   FinalColor.b *= ShadowingWeight.b;

   gl_FragColor = FinalColor;
}
```

Fig. 3-20: Shadow code for volume rendering with shadowing

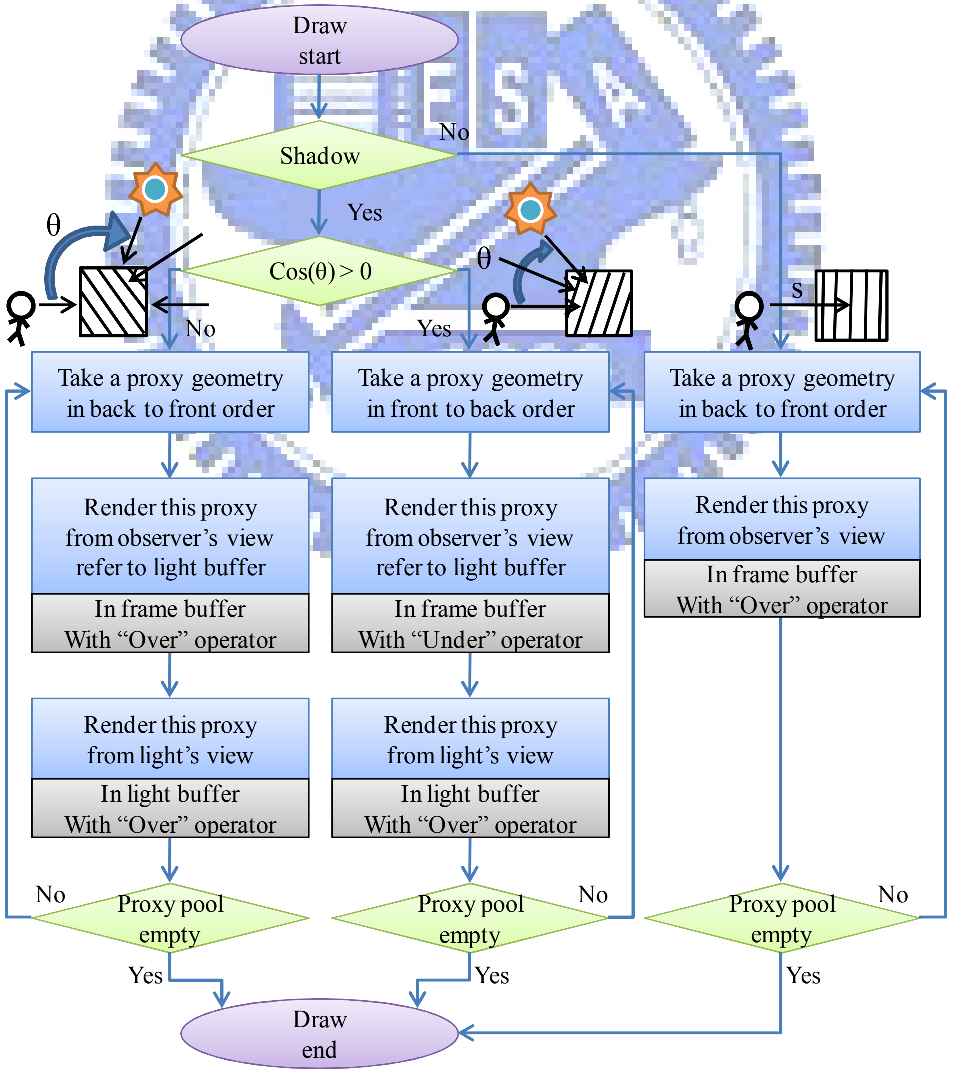

Draw start
Shadow
No
Yes
Cos(θ) > 0
θ
No
Yes
θ
S
Take a proxy geometry in back to front order
Take a proxy geometry in front to back order
Take a proxy geometry in back to front order
Render this proxy from observer's view refer to light buffer
In frame buffer With "Over" operator
Render this proxy from observer's view refer to light buffer
In frame buffer With "Under" operator
Render this proxy from observer's view
In frame buffer With "Over" operator
Render this proxy from light's view
In light buffer With "Over" operator
Render this proxy from light's view
In light buffer With "Over" operator
No
Proxy pool empty
Yes
Proxy pool empty
No
Yes
Proxy pool empty
No
Yes
Draw end

Fig. 3-21: Draw stage of main process

Chapter 4
Main Result

In this thesis, we build a volume data visualization system for confocal microscopic image of *Drosophila's* brain. Although there are some software platforms for visualization and manipulating bio-medical and science data in the market, user need to pay high price for authorization and these software are hard to accord with biologist's particular request. Our system provides high quality volume visualization to help biologist make profound diagnosis.

The implementation of our volume renderer is based on C++, OpenGL, and GLSL. The performance measurements were conducted on a Windows XP PC with an Intel E4400 CPU and an NVidia GeForce 8800GT graphics board with 1024MB texture memory.

Fig. 4-1 shows a fly brain of size 256*256*67. Fig. 4-1a is rendered by basic texture-based volume rendering with 60 frames per second (fps). Fig. 4-1b is rendered by pre-integrated volume rendering with 60 fps. Fig. 4-1c is rendered by pre-integrated volume rendering with lighting between 53 to 57 fps. Fig. 4-6d is rendered by pre-integrated volume rendering with shadowing between 13 to 18 fps. Fig. 4-6e is rendered by pre-integrated volume rendering with lighting and shadowing between 13 to 18 fps.

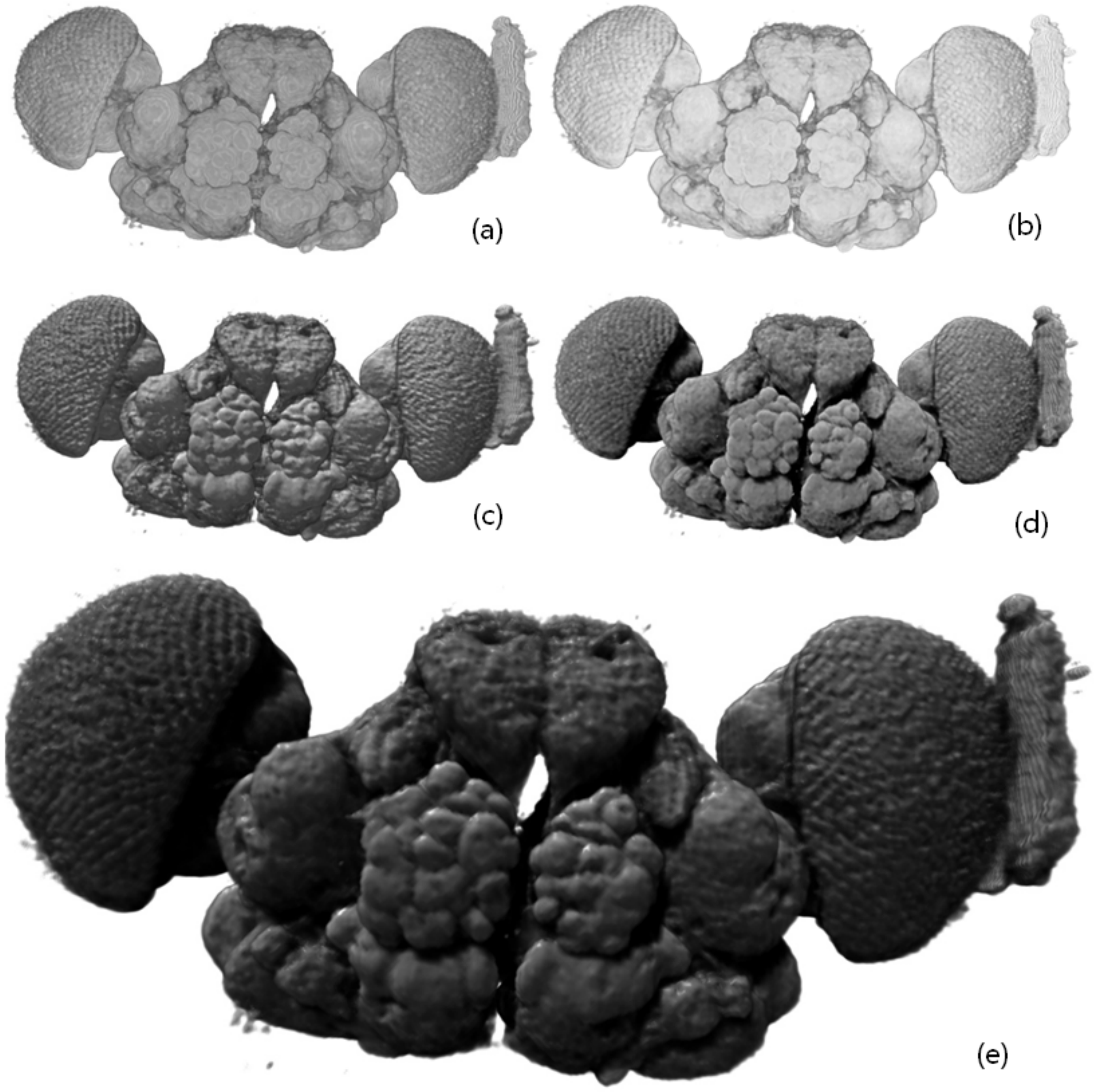


Fig. 4-1: Result of fly brain

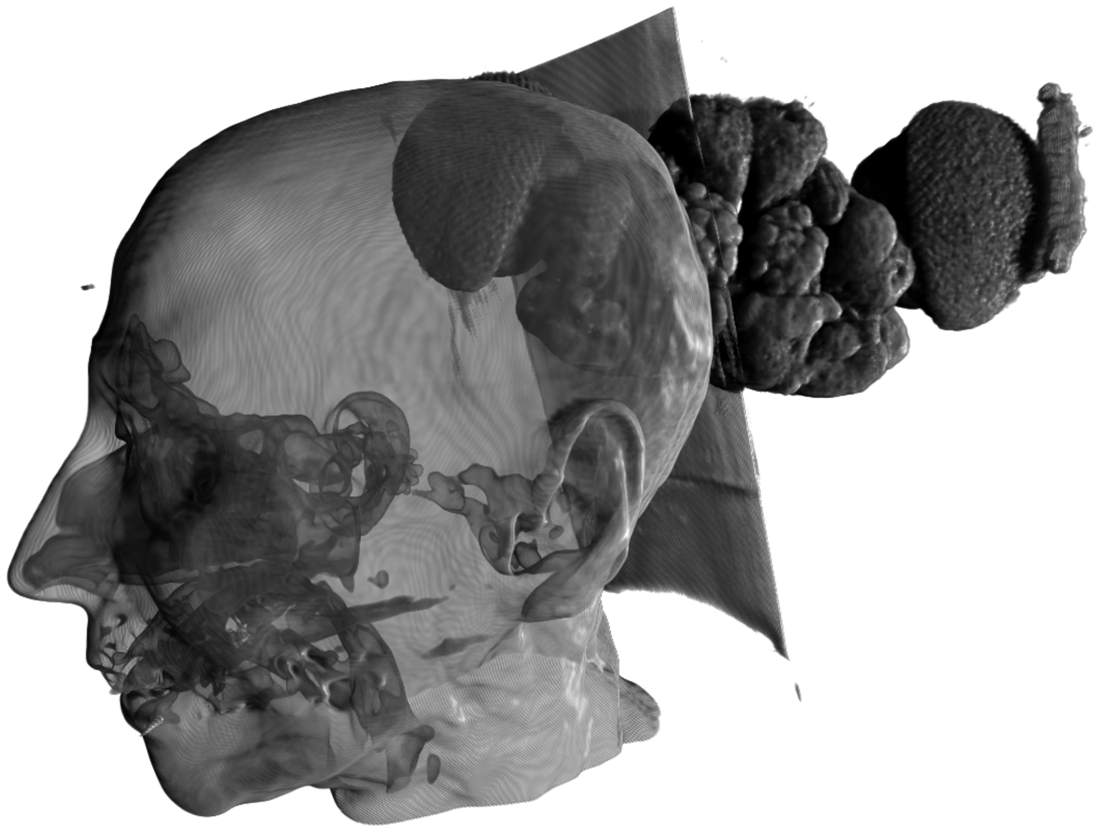


Fig. 4-2: Result of the multi-volume rendering algorithm

Fig. 4-2 shows the result of the multi-volume rendering algorithm. The left side of Fig. 4-2 is a human head of size 256*256*256 and the right side of Fig. 4-2 is a fly brain of size 256*256*67. As shown that the human head causes some shadows in the fly brain. The fps of Fig. 4-2 is between 7 to 10 fps.

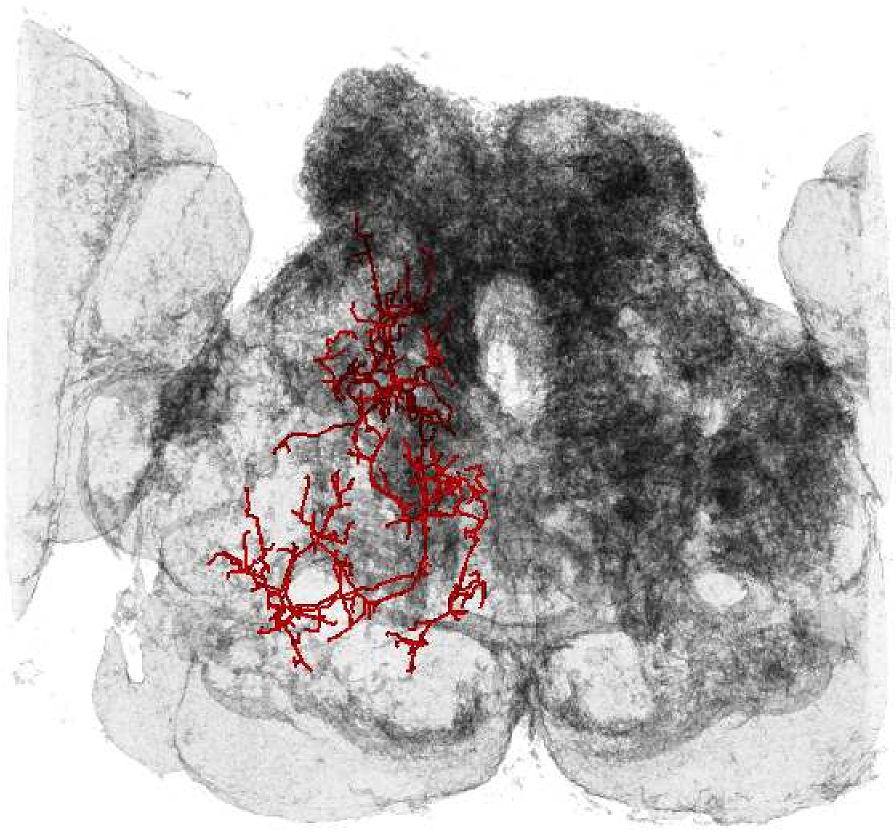


Fig. 4-3: A neuron in *Drosophila's* brain, glomerulus

Fig. 4-3 shows the result of texture-based volume rendering with geometry data. The fly brain of size 512*512*123 was rendered between 7 to 10 fps.

Chapter 5
Conclusion and Future Work

In this thesis, we build a visualization system for more reality and interactive volume rendering for confocal microscopic image of *Drosophila's* brain. We combine and reference many important researches to build this system, include improving low sampling rate by the pre-integrated volume rendering algorithm, providing reality volume rendering by lighting and shadowing, providing more space information by the volume rendering with geometry data and comparing between many volume data by the multi-volume rendering.

Although our system can provide interactive volume rendering with consumer graphics hardware, the performance is not good enough to provide good result with 30 fps. Our visualization system can be improved by referencing some accelerating texture-based volume rendering algorithm, like empty space skipping technique.

The data size of confocal microscopic image is usually larger than the texture memory of consumer graphic card. The volume rendering algorithm for large data is a important question. Our system can be improved by referencing some large volume data visualization algorithm, like level of detail or octree technique.

Although lighting and shadowing effect can improve the reality of volume rendering, the difference between materials cannot be observed. A shading system is needed to rendering more detail in different materials, like bone, skin, or tissue.

Bibliography

[1] A. V. Gelder, K. Kim, "Direct volume rendering via 3D texture mapping hardware," *Proc. of Vol. Rend. Symp. '96*, pp. 23-30, 1996.

[2] B. Cabral, N. Cam, J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," *Symp. on Volume Visualization '94*, pp.91-98, 1994.

[3] A. Kaufman, K. Mueller, "Overview of Volume Rendering," *The Visualization Handbook*, 2005.

[4] M. Ikits, J. Kniss, A. Lefohn, C. Hansen, "Volume Rendering Techniques," *GPU Gems, chapter 39*, pp. 667-692, 2004.

[5] K. Engel, M. Kraus, T. Ertl, "High-quality pre-integrated volume rendering using hardware-accelerated pixel shading," *SIGGRAPH/Eurographics Workshop on Graphics Hardware 2001*, pp. 9-16, 2001.

[6] B. T. Phong, "Illumination for computer generated pictures," *Communications of the ACM*, pp.311-317, 1975.

[7] A. V. Gelder, K. Kwansik, "Direct Volume Rendering with Shading via Three-Dimensional Textures," *ACM Symposium on Volume Visualization '96*, pp. 23-30, 1996.

[8] J. Kniss, G. Kindlmann, C. Hansen, “Interactive volume rendering using multidimensional transfer functions and direct manipulation widgets,” IEEE Visualization ’01, pp. 255-262, 2001.

[9] J. Kniss, S. Premoze, C. Hansen, P. Shirley, A. McPherson, “A model for volume lighting and modeling,” *IEEE Transactions on Visualization and Computer Graphics*, pp. 150-162, 2003.

[10] P.-C. Lee, Y.-T. Ching, H.-M. Chang, A.-S. Chiang, “A semi-automatic method for neuron centerline extraction in confocal microscopic image stack,” *ISBI 2008, 5th IEEE International Symposium on Biomedical Imaging*, pp. 959-962, 2008.

[11] N. Max, “Optical Models for Direct Volume Rendering,” *IEEE Transactions on Visualization and Computer Graphics*, Vol. 1, Issue 2, pp. 99-108, 1995.

[12] T. McReynolds, D. Blythe, B. Grantham, S. Nelson, “Advanced Graphics Programming Techniques Using OpenGL,” *SIGGRAPH’98 Course Notes*, 1998.

[13] E. Persson, “Framebuffer Objects,” ATI Technologies, Inc, 2005.

[14] T. Porter, T. Duff, “Compositing digital images,” Computer Graphics (Proc. Siggraph ‘84), pp. 253-259, 1984.

[15] R. J. Rost, “OpenGL Shading Language, 2nd edition,” *Addison Wesley*, 2006.