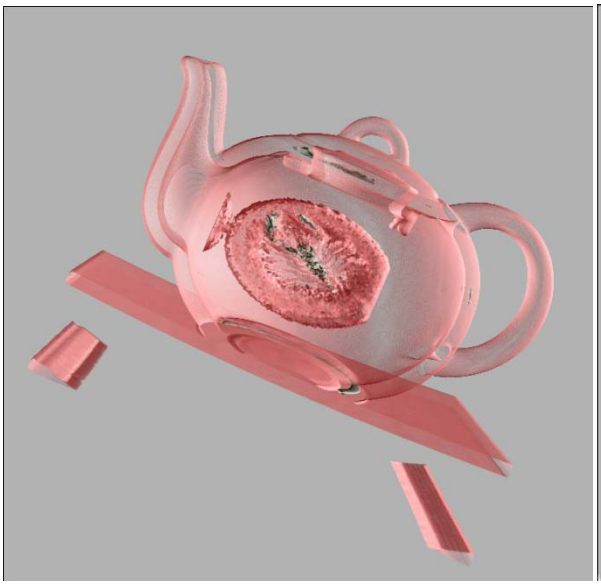
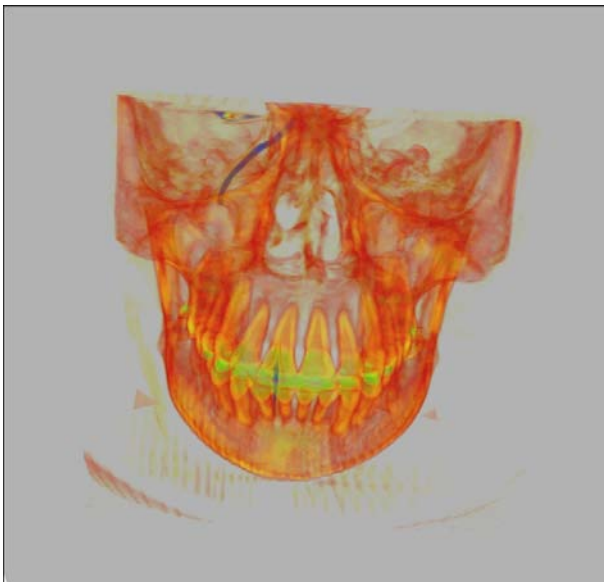
MASTER PROJECT THESIS

REAL-TIME VOLUME RENDERING FOR SCIENTIFIC VISUALIZATION

Min Jiang

MSc Computer Animation and Visual Effects

NCCA, Media School, Bournemouth University

18th August, 2011

Contents

CONTENTS II

ABSTRACT 2

1. INTRODUCTION 3

2. PREVIOUS WORK 4

3. THEORY 5

3.1 BASIC APPROACHES 5

3.2 VOLUME RENDERING INTEGRAL 6

3.3 TEXTURE SLICING 7

3.3.1 Object-Aligned Texture Slicing 8

3.3.2 View-Aligned Texture Slicing 11

3.3.3 Discussion 13

3.4 GPU-BASED RAY CASTING 14

4. IMPLEMENTATION 15

4.1 TEXTURE SET-UP 15

4.1.1 Texture Slicing Approach 17

4.1.2 Ray Casting Approach 17

4.2 GEOMETRY SET-UP 18

4.2.1 World Space, View Space, and Texture Space 18

4.2.2 Object-Aligned Slicing Method 19

4.2.3 View-Aligned Slicing Method 22

4.2.4 Ray Casting Method 24

4.3 TEXTURE MAPPING 26

4.3 CLASSIFICATION 27

4.4 LOCAL ILLUMINATION MODELS 29
4.4.1 Blinn-Phong Illumination 30
4.4.2 Gradient-Based Illumination 32
4.4.3 Gradient Estimation 32
4.5 COMPOSITING 34

5. RESULT AND ANALYSIS 36

5.1 OBJECT-ALIGNED SLICING METHOD 36
5.2 VIEW-ALIGNED SLICING METHOD 39
5.3 RAY CASTING 44
5.4 EFFICIENCY COMPARISON 46

6. CONCLUSION 47

7. BIBLIOGRAPHY 49

Abstract

Volume rendering techniques allow high quality visualization of volumetric data sets, and have so far been widely used in the realm of medicine, geosciences and engineering. This project presents a framework for GPU-based direct volume rendering methods. Both texture slicing and ray casting methods will be discussed. For slice-based methods, the predominant proxy geometries are either view-aligned or object-aligned slices. Since the object-aligned technique causes visible artifacts, due to different adjacent sampling distances from perspective view angle, a view-aligned technique has been implemented to achieve a better result. With the increasing pace on the evolution of graphic hardware, a GPU-based ray casting method has been developed to meet the criteria for an ideal volume rendering. The volume shaders implemented for ray casting method in this work are single-pass ray casting.

Key words: volume rendering, slice-based, object-aligned, view-aligned, ray casting, transfer function

1. Introduction

Volume visualization plays an important role in both academic and industry domains. In addition to modeling and rendering volumetric phenomena, such as fluids, clouds, smoke, fire and dust, volume rendering is essential to scientific and engineering applications that are required to measure or generate 2-D projections of a colored semitransparent volume by sampled functions (Pawasauskas 1997). Medical scans (CT, MRI), 3D photography and mechanical simulation are all typical examples of the complex volume data which need to be visualized, stored, or transmitted (Lee Desbrum Schroder 2003).

In general, there are two different kinds of volume visualization techniques. One is visualizing volumetric data by extracting iso-surfaces from discrete three-dimensional data, called **indirect volume rendering**, such as the classical marching cubes algorithm (Lorensen and Cline 1987). The main disadvantage of this method is that the use of surface configurations of cubes causes wrong surface production and hole generation (Jin et al 2006). Also, for high resolution volume data the number of generated triangles can be extremely high, thus the computation cost is quite expensive. The other kind of visualization technique is **direct volume rendering**, where algorithms map each voxel (an individual volume element, as pixel for “picture element”) to optical properties, such as color, opacity and gradient vector, rather than dealing with geometric surfaces from the

volume. Direct volume rendering has proven to be more robust and flexible than the indirect ones in visualization methods for three-dimensional scalar fields (Engel et al 2006). In this project, we will focus on the direct rendering methods.

2. Previous work

Since the basic principle of volume rendering was introduced by Kajiya (1984), most of the work in direct volume visualization were driven by the evolution of new graphic processors. The RealityEngine graphics system led to the establishment of the so-called slicing methods (Akeley 1993). Subsequently, application on PixelFlow graphics system overcomes the limitation imposed by RealityEngine, accelerating the interactive frame rates (Cullip and Neumann 1993). Ray casting has been a well-known method for CPU-based volume rendering since the 1980s (Levoy 1988). However, GPU- based ray casting is still quite a new exploration with its first implementation in 2003 (Rottger et al 2003). The reason for the late development of GPU-based ray casting is that the advanced fragment shader functionality was not available earlier (Kajiya 1984). With more advanced texture mapping capabilities of today's graphics hardware, many enhancements to the ray casting principle have been proposed to “increase the interactivity and applicability of the method” (Weiler et al 2003). Also, some acceleration techniques proposed for the original ray casting approach, such as early ray termination and empty space skipping have been successfully adopted to

texture-based direct volume rendering (Kruger and Westermann 2003) (Li, Mueller and Kaufman 2003) .

3. Theory

3.1 Basic Approaches

Direct volume rendering can be further classified as either image-order or object-order methods (Engel et al 2006). Image-order approaches generate the final image pixel-by-pixel casting a ray from each pixel, and re-sampling the volume along each ray at evenly located sample points, the most popular image-based method is *ray casting*. On the other hand, object-order methods follow a certain organized scheme to scan the 3D volume voxel by voxel in its object space. The traversed volume areas are then projected onto the image plane. The typical example of the object-order method is *texture slicing*, which is also a dominant technique for GPU-based volume rendering. There are other object-order methods such as *shear-warp volume rendering* (Csebfalvi Konig and Groller 2000) , *splatting* (Westover 1990) and *cell projection* (Shirley and Tuchman 1990).

In the following sections we will mainly discuss the ray-casting and texture slicing methods. They are both GPU-based direct volume rendering approaches.

3.2 Volume Rendering Integral

Direct volume rendering by its nature deals with transparent objects. Before rendering a translucent volume, we need to know the interaction between light and volume, including absorption, scattering, or emission. When the light is passing through the volume, the interaction needs to be evaluated at all positions in the 3D volume.

An *optical model* is used to describe the behaviors of light transportation (Moreland 2004). During rendering, the optical model assigns optical properties, such as color and opacity, to each voxel. The most commonly used model is the emission-absorption optical model:

$$I(D) = I_0 e^{-\int_{S_0}^{D} k(t)dt} + \int_{S_0}^{D} q(s)\, e^{-\int_{s}^{D} k(t)dt} ds \qquad (3.1)$$

In which, k is the absorption coefficient and q describes the emission. The integration shows the light information from the entry point S_0 to the exit point D. More complex one contains shadows and illuminations, which account for light scattering effects. (Engel et al 2006)

For discrete volume rendering, each voxel corresponds to a position in the data space. The optical properties are accumulated along each viewing ray to form a projection of the 3D volume data. The accumulated color and opacity are computed according to the discrete volume rendering equation:

$$C = \sum_{i=1}^{n} C_i \prod_{j=1}^{i-1}(1 - A_j) \qquad (3.2)$$

$$A = 1 - \prod_{j=1}^{n}(1 - A_j)$$

In which, C_i and A_i are the color and opacity of the voxel at sample i. Opacity A_i evaluates the absorption, while color C_i approximates the emission which is opacity-weighted by A_i. Because Equation (3.2) is a numerical approximation to the continuous optical model as Equation (3.1), the *sampling rate*, the length of ray segment between sample *i* and sample *i+1*, has a great influence on the "accuracy of the approximation and the quality of the rendering" (Fernando 2004).

3.3 Texture Slicing

Texture slicing techniques approximate the volume rendering integral by rendering a stack of geometric primitives inside the volume. These geometric primitives (usually polygonal slices) only represent a *proxy geometry*, as shown in Figure 3.1. They only describe the shape of the data domain, usually the bounding box, not the shape of the object contained in the data.

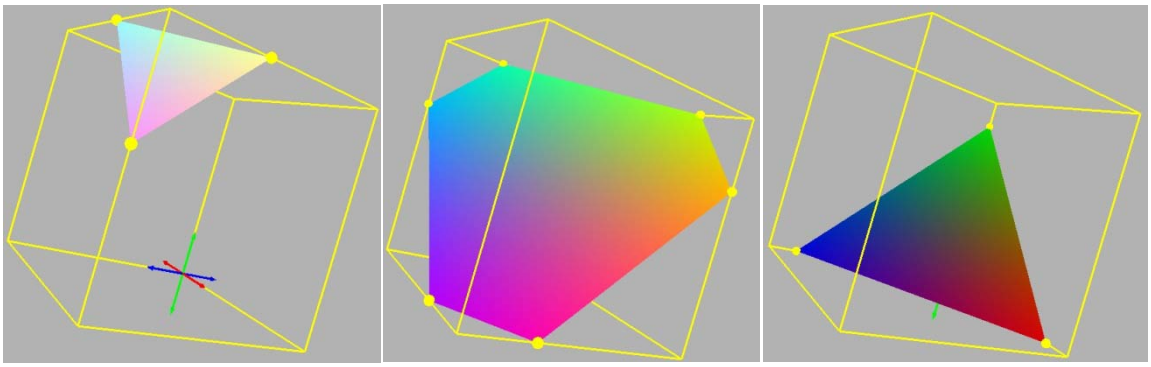


Figure 3.1. Proxy geometry

The proxy geometry is rasterized and blended into the frame buffer in back-to

front or front-to-back order. This process is called *compositing*, which will be looked at in detail in Section 4. In fragment shader, the calculated texture coordinate are used as a texture lookup. Each geometric primitive is assigned texture coordinates for sampling the volume texture. By projecting a high number of semi-transparent slices onto the image plane according to the compositing scheme, a 3D data set can be visualized. Texture slicing is widely used and very efficient because it only needs texture support and blending. According to the direction of the slices, it can be further categorized into Object-aligned texture slicing and View-aligned texture slicing methods.

3.3.1 Object-Aligned Texture Slicing

In Object-aligned slicing method (also referred as axis-aligned slices), the proxy geometry are oriented along with one of the major axes in object space.

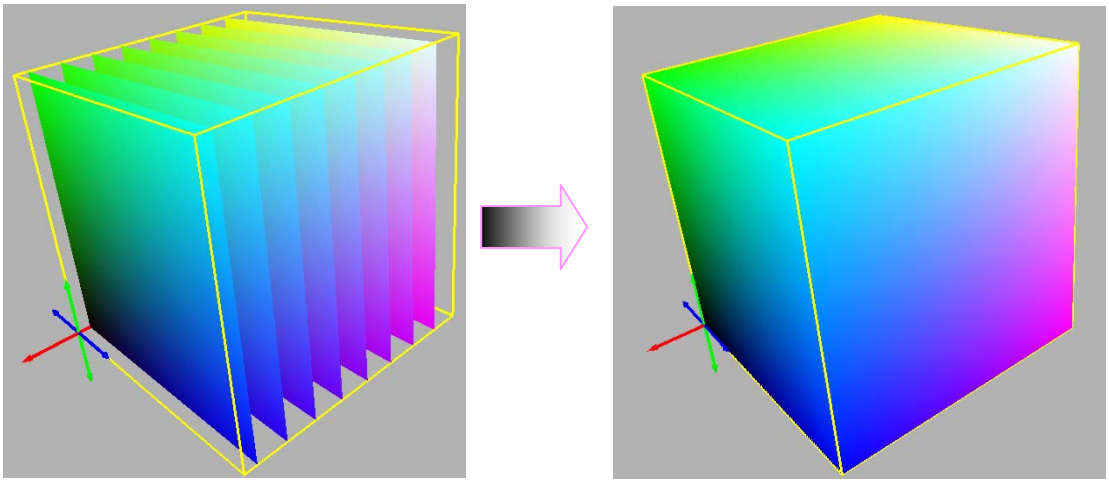


Figure 3.2. Object-aligned slices used as proxy geometry

At the beginning of the implementation, volume data is loaded into OpenGL and

volume texture. The proxy geometry are initialized as six stacks of slices, respectively aligning with six axis (including positive and negative). After initialization, the polygonal slices are mapped with the respective 3D texture according to the texture coordinates. To allow the interactive rotation of the volume data, every time the viewing angle changes, the direction of the proxy geometry must be chosen again. The major axis must be selected “in a way that minimizes the angle between the slice normal and the viewing direction” (Engel et al 2006). This will effectively avoid viewing slices parallel to the viewing angle, which will results in none sample while intersecting. Therefore, with an angle larger than 45^{o}, the stacks must be switched. When the angle between viewing direction and the slice normal is 45^{o}, the slicing direction will become ambiguous and can be chosen arbitrary.

The main advantages of the object-aligned slicing method are its simple concept and its high speed performance. However, it comes with several drawbacks. The sampling rate is depended on the distance between two slices. Once the distance is fixed, the sampling is fixed. It is easy to assume that the sampling points along the viewing direction have a fixed distance, resulting in a constant sampling rate. However, when applying a perspective projection, the distance between adjacent sampling points depends on the angle at which the assumed viewing ray intersects the slices (see Figure 3.3).

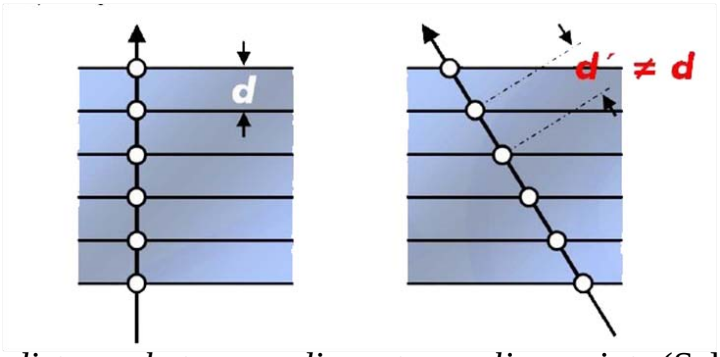


Figure 3.3.The distance between adjacent sampling points (Salama 2006)

As a consequence, the result will only be accurate when the direction of viewing ray is perpendicular to the slices. The varying sampling distance with the view angle leads to visible artifacts, as shown in Figure 3.4. This shows two different sampling results due to slightly different viewing angles. We can increase the sampling rate to alleviate the artifacts and improve the image quality, but it doesn't solve the problem fundamentally, and also, the higher sampling rate is at the cost of the computational complexity. In addition to the sampling artifact, the abrupt change of the stacks at 45^{o} results in flicking because the sharply shift of the sampling positions.

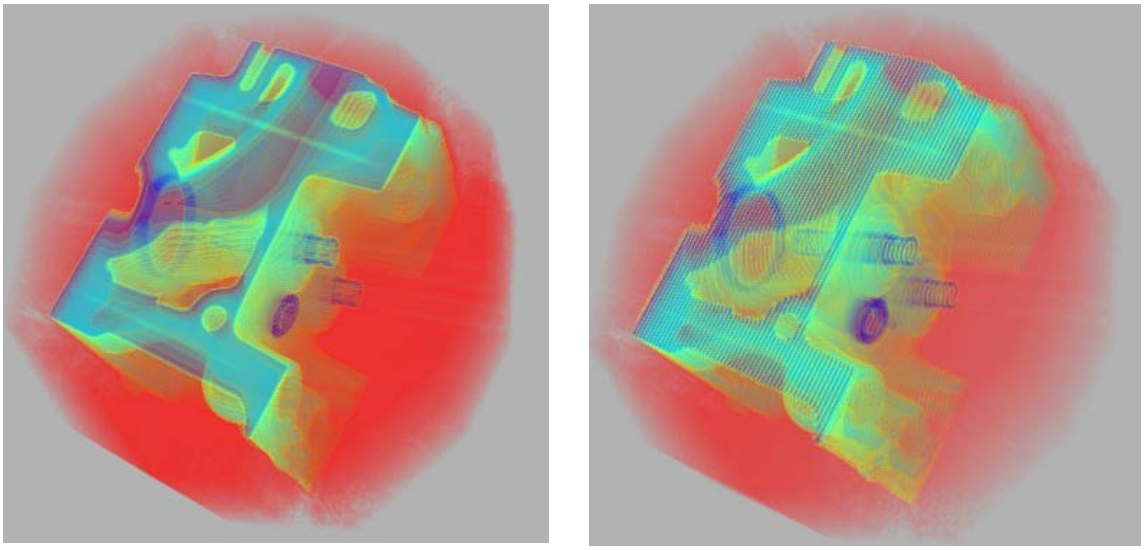


Figure 3.4. Aliasing artifacts of object-aligned slicing method

Due to the limited options available to improve the method, we turn to the view-aligned slicing method which will completely circumvent this problem.

3.3.2 View-Aligned Texture Slicing

The view-aligned slicing method achieves a more consistent sampling rate for different viewing directions by using viewport-aligned slices. This means the volumetric object is cut into slices orthogonal to the view direction, shown as Figure 3.5. The proxy geometry must be recomputed as long as the viewing direction changes. Because there is no abrupt change of slicing stacks, the flickering artifacts have been removed.

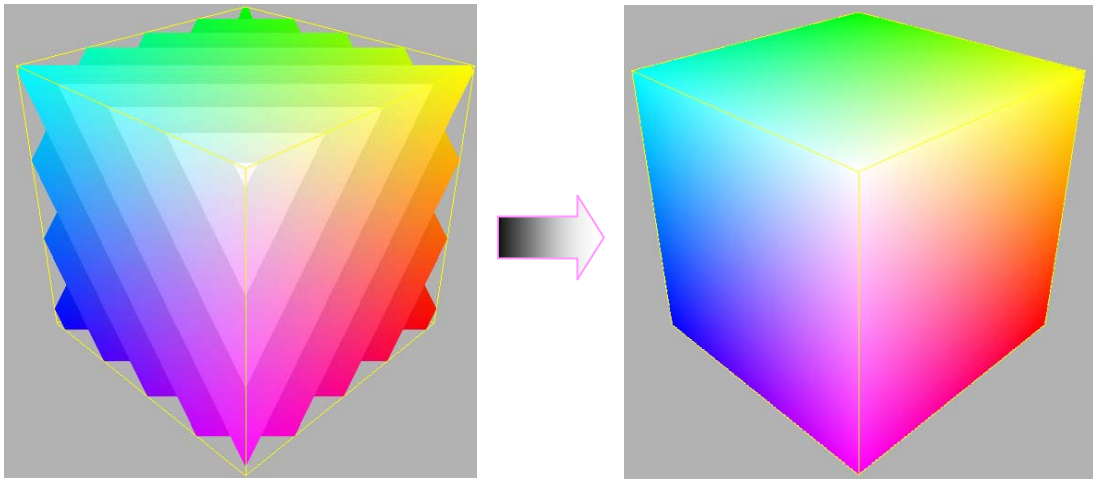


Figure 3.5.View-aligned slices used as proxy geometry

In parallel projection, the consistent sampling rate is been guaranteed (see Figure 3.6 (a) and (b)). In the case of perspective projection, the distance of sampling points varies a bit (see Figure 3.6 (c)). However, it is only noticeable if the field of view is extremely large. We will come across the same problem when we are

doing ray casting. As long as the virtual camera views the volume object from a certain far distance, the effect of the inconsistent sampling rate is hardly visible.

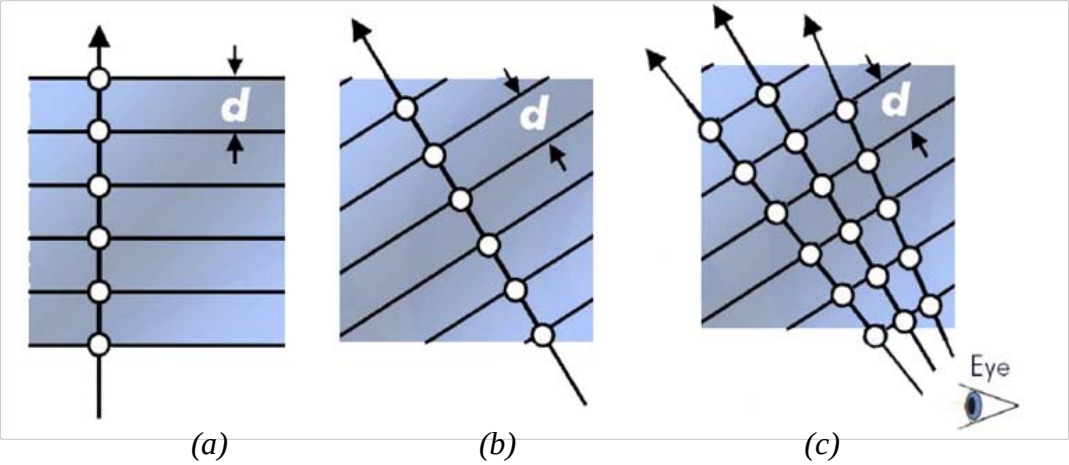


Figure 3.6.Sampling distance of view-aligned slices (Salama 2006)

View aligned slices are first drawn in view space and subsequently the corresponding coordinates are calculated in the world space. By intersecting the slices with the volume bounding box, we can decide which parts of these slices are within the texture bounding box. The outside parts will be discarded in the fragment shader, considering we can't add or remove any vertex in vertex shader. For each pixel of the slices, the corresponding 3D texture coordinate is calculated in fragment shader, and the color and opacity will be assigned by the texture.

Compared with the object-aligned slicing approach, the view-aligned slicing method has proved superior in term of image quality, "removing some of the significant drawbacks while preserving almost all the benefits" (Engel et al 2006).

3.3.3 Discussion

Texture slicing volume rendering techniques owe their success and popularity to the fast sampling and high rasterization performance of hardware, and moreover the basic rendering principle are quite simple to understand and implement.

However, texture slicing volume rendering has a number of significant drawbacks, especially for large volume data sets. Since the number and the position of slices directly determine the rendering quality, the texture slicing approach will be strongly influenced by the complexity of the data sets. While in volume rendering, CT or MRI scan could be extremely large and in general it is common that a significant number of fragments does not contribute to the final image because are complete transparent or invisible. Furthermore, most volume rendering applications only focus on visualizing boundaries of objects or selected interesting regions. Therefore, texture slicing approaches are “rasterization-limited and can be hardly optimized from algorithmic point of view”. (Stegmaier et al 2005)

On the other hand, the evolution of the modern programmable GPU and associated interfaces such as OpenGL, have led to novel graphics processors which provide an ideal platform for efficient ray casting implementations for volume rendering. The fragment shader of ray casting does not suffer from the flexibility issues. Furthermore, advanced fragment program promises a faster functionality that attracts lots of hardware manufacturers, which guarantee the advance of the

technique. According to these advantages, ray casting can be assumed as a future-proof approach for volume visualization.

3.4 GPU-based Ray Casting

The basic ray casting concept is to trace the rays from the camera to the volume and evaluate the volume rendering integral along the rays. The main advantage is that these rays are traced independently from each other through the volume. This gives more flexibility for implementing optimization strategies, such as “early-ray termination, adaptive sampling, and empty space skipping.” (Engel et al 2006)

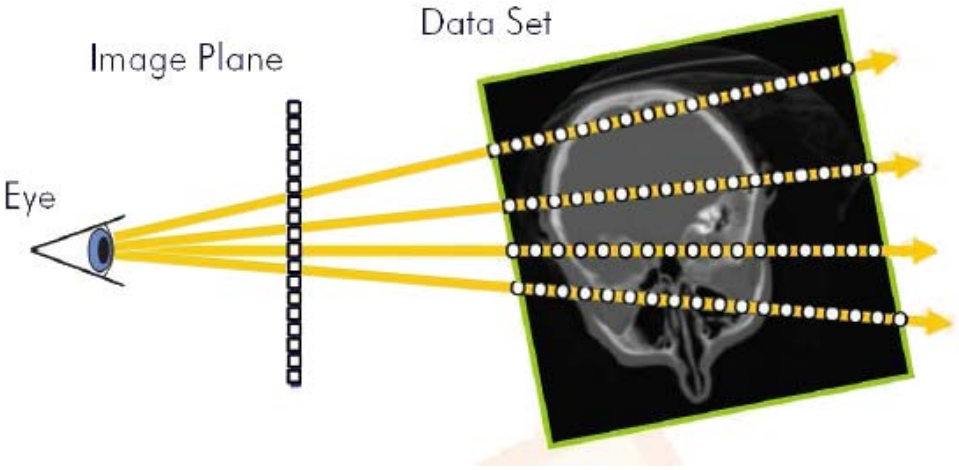


Figure 3.7. Ray casting principle (Salama 2006)

In this project, single-pass GPU ray casting has been implemented. For each pixel of the bounding box of the volume data, a single ray is built from the camera. The volume data is re-sampled at discrete positions along the ray. The scalar values of the volume data are mapped to optical properties by accumulating light information along the ray. The ray-casting algorithm can be described by the

pseudo code in Listing 3.1. The detail of implementation will be presented in the next section.

```
Compute volume entry position
Compute ray direction
While the ray position is in the volume
    Lookup the data value at current position
    Compositing the optical properties
    Update the position along the ray
End while
Draw pixel colour
```

Listing 3.1. pseudo code for ray casting

4. Implementation

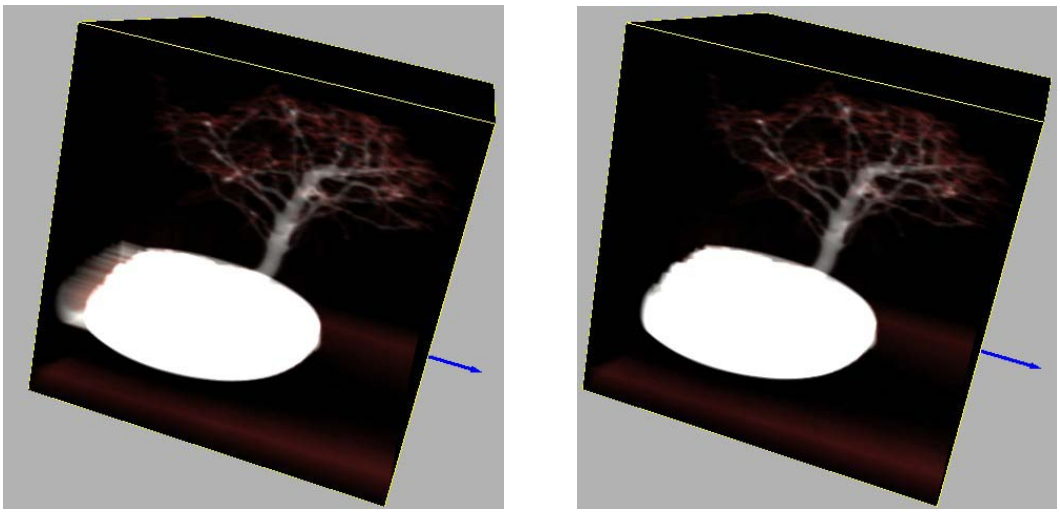
The implementation of volume rendering in this project consists of three major parts: a framework for the texture slicing approach, including both object-aligned slicing and view-aligned slicing methods, and a framework for the ray casting approach. They are all written in C++ with Qt and based on OpenGL and NGL libraries. The implementation of each framework has been introduced based on the volume rendering pipeline in the following sections.

4.1 Texture Set-up

A number of volume data sets have been implemented to illustrate the flexibility of the framework, such as Foot, Teapot, Engine, Skull, etc. The dataset library is

courtesy of University of Tübingen (2005). All the datasets are binary, storing 8 bit voxels for all slices and voxels in *raw* format.

First, we need to load the volume data into a 3D texture. Above all, the data is loaded from the raw file into an array buffer, then the array is passed into local graphics memory using *glTexImage3D* function. The internal format is set to GL_INTENSITY, which means that the emission or absorption values are stored as an intensity value for each voxel.



(a) *(b)*

Figure 4.1.texture loading

One thing that needs to be noted here is that if the first and the last slice contain data, we need to clear them with 0, otherwise this might cause problems when we render the data using ray casting method, shown as Figure 4.1. The tree in Figure 4.1 (a) seems “leak out” from the cube, after clearing the end slices, the problem has been fixed as Figure 4.1 (b).

In this project, all the texture datasets are loaded when OpenGL is initialized, including the transfer function texture, which will specified later. Therefore, when the texture data is passed into graphics card, we need to tell the GPU which texture is used as the volume one, and which is for transfer function. In this case, *glActiveTexture* function is used to control which texture unit will affect. Each texture unit is assigned with a texture ID and *glActiveTexture (textureID)* selects the active texture unit. The number of texture units available is hardware dependent.

4.1.1 Texture Slicing Approach

For the texture slicing approach, the textures are prepared for stacks of slices. During rasterization, each slice is textured with the optical properties directly from its corresponding 3D texture map.

4.1.2 Ray Casting Approach

For the ray casting approach, the texture is mapped with the ray samples within the bounding box of volume data rather than stacks of slices. The ray is just a virtual beam as a visual cue intersecting with objects, so it deals with non-planar surfaces or solids.

4.2 Geometry Set-up

This component mainly corresponds to the data-traversal step, which builds up the polygonal slices or ray samples to perform the intersection with the volume data and determines the sampling positions.

4.2.1 World Space, View Space, and Texture Space

Before talking about setting up the geometry, we need to specify the three different spaces which will be used in the following sections. Figure 4.2 shows the relationship between each other. Once an object has been created, each vertex of the object will have a relative coordinate to its centre in *Object Space*, and also, a relative position to the origin of the world, which is obtained by multiplying the *Model Matrix* with the model space coordinates. Therefore, the *Model Matrix* contains the object transformation in the *World Space*. To see the object, a virtual camera is introduced into the world, which gives the object a relative position to the camera, called *View Space*, which is computed through *View Matrix*. Therefore, all the object and camera transformations are stored in *ModelView Matrix*. It presents the transformation from the World Space coordinate system of the volume object into the View Space. The *Projection Matrix* is used to get the projection of the object onto the image plane.

The *Texture Space* is used at the texture-mapping stage. To bind the volume

texture with the object, we need to know the texture coordinate of each vertex to get the color and opacity information for the vertex of the object.

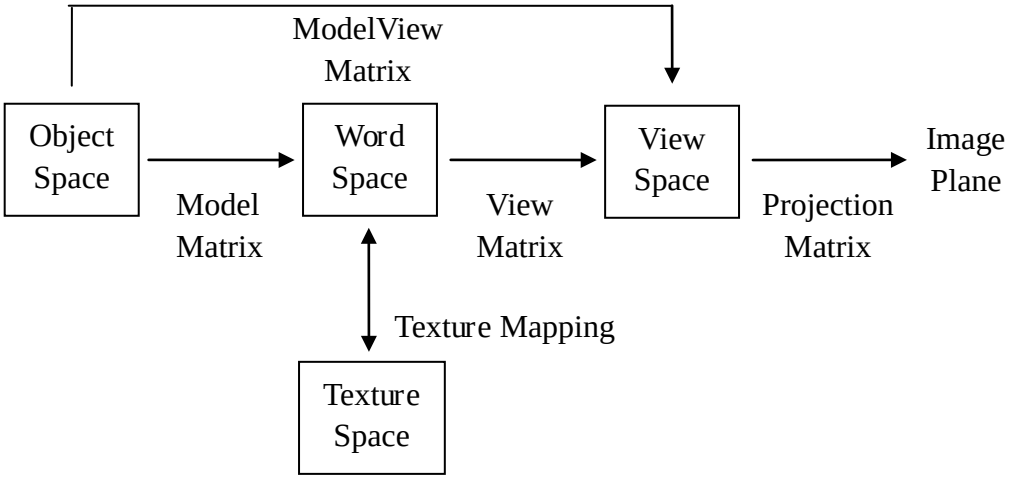


Figure 4.2.Relationship between spaces

4.2.2 Object-Aligned Slicing Method

In object-aligned slicing method we pre-computed and stored six stacks of slices along each axis in a bounding box. The bounding box confined the number of slices and their distances. These slices were a high number of equidistant planes which were equally-sized and equally-oriented in each direction. When we were drawing the polygonal slices, we need to choose the major axis to minimize the angle between the slice normal and the viewing direction.

Axis selection: To calculate the viewing direction relative to the volume object, the *ModelView Matrix* must be obtained from the current OpenGL state. The viewing direction of the camera in *Object Space* is originally in the negative z-axis direction. It needs to be transformed into *World Space* by multiplying the

ModelView Matrix. The corresponding stack of slices is then chosen according to the maximum component of the transformed viewing vector.

To show the slice movement with the corresponding texture, the intersections of the slice and the bounding box have been displayed. To calculate the position of the intersections, we performed the ray-plane intersection algorithm. In this approach, the slice is defined by three vertices, and the bounding box of the volume texture is regarded as a combination of 12 edges. To find the intersections, we basically repeated the ray-plane intersection algorithm for 12 times. The method is inspired by Sunday (2001), but much simpler and more efficient for this project.

Ray-Plane Intersection Assume a ray from P_0 to P_1, which are the two endpoints of each edge. Three vertices V_0, V_1 and V_2 form a plane. The normal vector of the plane $\vec{n}$ can be computed by the cross product of two vectors within the plane.

$$\vec{n} = (V_2 - V_0) \times (V_1 - V_0) \qquad (4.1)$$

Then, we calculate the direction of the ray, which is defined as:

$$\vec{p} = P_1 - P_0 \qquad (4.2)$$

If $\vec{n} \cdot \vec{p} = 0$, which mean $\vec{n}$ and $\vec{p}$ are perpendicular, the direction of the ray is parallel to the plane, thus there is apparently no intersection between the ray and plane.

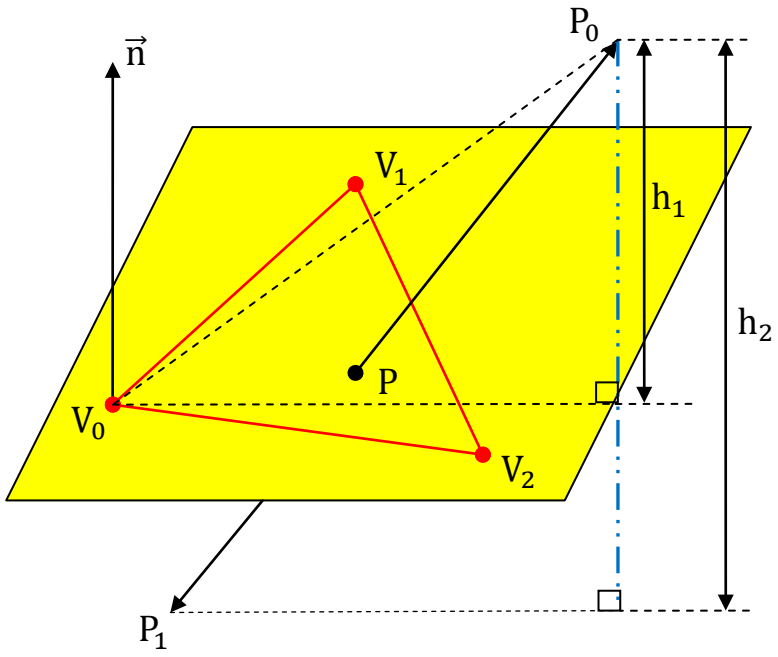


Figure 4.3. Ray-Plane Intersection algorithm

Another situation we need to exclude is that they do have a intersection, but it is at the extension line of the ray, which means it already exceeds the length of the edge. Assume the intersection is P, shown as Figure 4.3. We first calculate the dot product of the ray and the plane normal, which is the projection of $\overrightarrow{P_0P_1}$ on $\vec{n}$, then we need the projection of $\overrightarrow{P_0P}$ on $\vec{n}$. By comparing these two projections, we get the ratio of these two vectors. If the ratio is between 0 and 1, it means P is along the ray from P_0 to P_1. However, we don't know the position of P. In fact, the vector can be alternately obtained from computing the projection of $\overrightarrow{P_0V_0}$ on $\vec{n}$, because both V_0 and P are within the plane. They share the height from one point to the plane. Finally, the position of P can be easily obtained as Equation 4.4. The Ray-Plane Intersection algorithm is described by the pseudo code in Listing 4.1.

$$h_2 = \vec{p} \cdot \vec{n} \qquad h_1 = \overrightarrow{P_0V_0} \cdot \vec{n} \qquad ratio = h_1/h_2 \qquad (4.3)$$

$$P = P_0 + \overrightarrow{P_0P_1} \cdot ratio \qquad (4.4)$$

```
Compute the normal of the plane
Compute the direction of the ray
If the direction of the ray is perpendicular to the normal
    No intersection
Computer two projections
If the ratio>1 or ratio<0
    The intersection exceeds the length of the edge
Else
    Compute the position of the intersection
```

Listing 4.1. pseudo code for Ray-Plane Intersection algorithm

4.2.3 View-Aligned Slicing Method

For the view-aligned slicing method, the geometry set-up component created view-aligned slices. It is assumed that the intersection calculation was performed on the GPU because the slices were originally drawn in the *View Space*.

Calculating the *World Space* coordinates of view-aligned planes in the bounding box is a more complicated task. Polygonal slices were first drawn along the view direction in *View Space* (See Figure 4.4). Therefore, in vertex shader, the vertex position was obtained by only multiplying with the *Projection Matrix*. In fragment shader, we derived this proxy geometry by projecting the vertexes back to the *World Space* using the inverse of the *ModelView Matrix.* Consequently, the resulting quadrilateral is viewport-aligned.

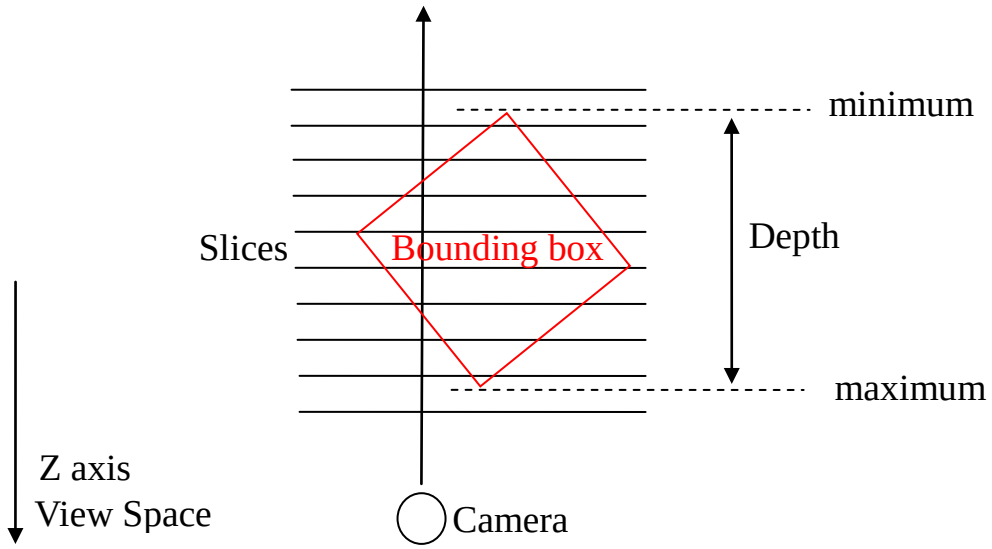


Figure 4.4. Depth Calculation for view-aligned slicing

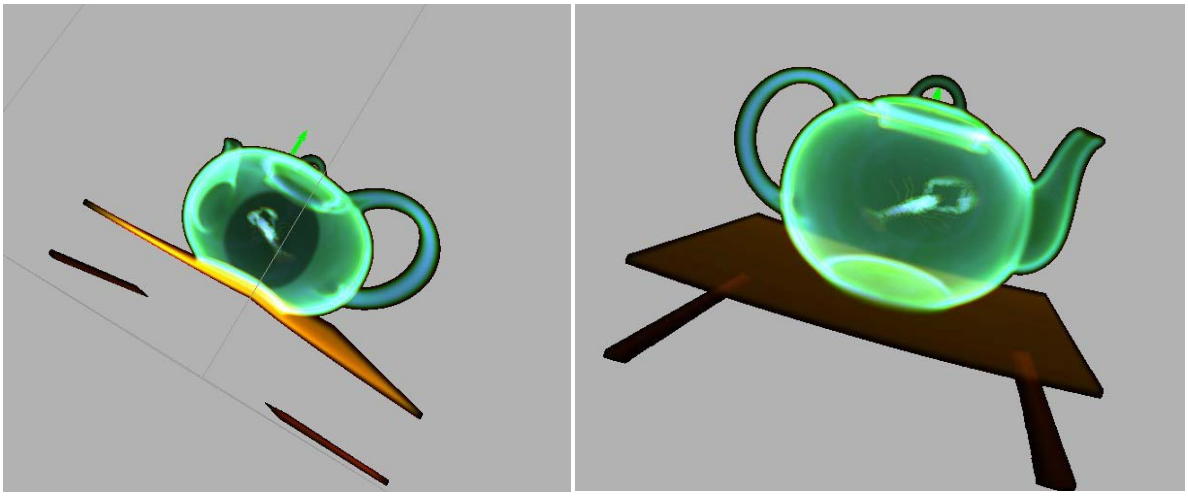
Depth computation: The z-axis coordinate of the slices in View Space were specified by a uniform parameter *depth* in vertex shader, which was updated for each slice to be rendered. Because the slices were at first drawn in the View Space, so we need to calculate the corresponding range of depth to make sure all the volume data in the bounding has been covered. The size of the bounding box was controlled by two uniform parameters *mMin* and *mMax* passed from CPU. In our project, all the bounding box was set as a unit cube with the centre in the origin. The depth of the bounding box from the camera was calculated by multiplying the eight vertices with the ModelView Matrix. The minimum and maximum of the eight z-axis coordinates comprised the depth of the slices. We did that in CPU to avoid calculating the range for each vertex every time. The depth needed to be updated when the *ModelView Matrix* changed.

4.2.4 Ray Casting Method

For ray casting, the geometry set-up is mainly concerned with the parametric set-up for the ray traversal. All the intersection calculation is performed in the fragment shader.

Camera Position is used to calculate the ray direction as the starting point of the ray. Since the camera is initially located at the origin in *Object Space*, the camera position in *World Space* is computed by reversing the transformation dictated by the *ModelView Matrix*.

The camera position needs to be set in a distance from the object, otherwise, it will cause the inconsistent sampling rate mentioned by Section 3.3.2, which will then lead to a distortion to the rendering object. To move the camera far away, then increase the scale of the transformation stack, it will solve the problem.



(a) Distorted teapot *(b) teapot without distortion*

Figure 4.5. Distortion caused by the camera position

Ray Direction is calculated afterwards. The ray's entry point to the volume data is given by the vertex coordinate of the bounding box. We can get the vertex positions also in *World Space,* therefore, we calculated the ray direction in the World Space. As a consequence, the ray direction is a uniform vector from the camera position to the entry point.

It is now possible to sample the volume data with a set of loop, allowing for an overall number of 1024 iterations in this project. Because this process was related to the volume texture and took part in *Texture Space*, the ray's entry point used here and all the texture sampling positions are the interpolated texture coordinates.

Loop: The ray evaluates the volume rendering integral when it traverses through the volume data. The ray is sampled at discrete positions, and the traversal loop scans the rays along these positions. For each iteration of the loop, the current sample point performs the following subtasks: texture mapping, classification and compositing. More information will be described in the following sections. Subsequently, the current ray position is advanced to the next sampling location along the ray by a specific step size.

Ray Termination: The traversal loop ends when the ray leaves the volume dataset. Only when the current ray position is still in the volume, it enters next loop. The texture coordinates are between (0,0,0) and (1,1,1). If the current coordinate is out

of this range, breaks out of the loop.

4.3 Texture Mapping

Texture mapping determines the sampling position and texture coordinates of the vertices that need to be rendered. The operations basically interpolate or filter a volume texture to obtain the color samples at specific location. The color samples are usually scalar values between 0 and 1.

Volume rendering assumes a continuous 3D scalar field, which can be written as a mapping

$$\emptyset: R^3 \rightarrow R \qquad (4.5)$$

which is a function from 3D space to a single-component value (Fernando 2004). In fragment shader, we use the simple function *texture3D(VolumeTexture, TexCoordinate)* to get the sample color from the active 3D texture.

To get the texture coordinates, the world spatial position within the bounding box is transformed into the *Texture Space*. We use the minimum and the maximum vertices to denote the range of the bounding box in the World Space. To remap it into the texture coordinates range from 0 to 1, we need to do an interpolation as

$$\text{TexCoordinate} = \frac{\text{worldPos} - \text{min}}{\text{max} - \text{min}} \qquad (4.6)$$

Before assigning color sample to each vertex, we can decide if the vertex is going to be rendered or not using the fragment shader keyword ***discard***. It terminates the shader for the current fragment without writing to the frame buffer or depth (Fernandes 2011). Hence, we can only render the fragment we are interested in. In this project, we used *discard* in several ways. Firstly, we discarded the fragment when the texture coordinate is not between (0,0,0) and (1,1,1). Secondly, we discarded the fragment when the ray did not hit anything we were interested in. Thirdly, we discarded the transparent fragment of the volume data.

4.3 Classification

Volume data set contains abstract scalar data values that represent some spatially varying physical property, such as density, temperature, or strength. In general, there is no natural way to obtain emission and absorption coefficients from such data. Instead, the user needs to decide how the different structures in the volume data should look by mapping locally measured data properties to optical properties. This mapping method is called ***transfer function***. Transfer functions are essential to direct volume rendering because they make the data visible. “Good transfer functions reveal the important structures in the data without obscuring them with unimportant regions” (Kniss Kindlmann and Hansen 2002). “The process of finding an appropriate transfer function is often referred as ***classification***.” (Engel et al 2006)

Transfer function design is a difficult and tedious task. It requires significant insight into the underlying data set. The feature of interest is not easy to identify in the transfer function domain. Moreover, it is difficult to isolate the interesting regions because other regions may share the same range of the data values (Kniss Kindlmann and Hansen 2002). The simplest and most common transfer functions are one dimensional, and they assign color and opacity to the voxel data (Fernando 2004). Typically, transfer functions are implemented with 1D texture lookup tables.

The first project illustrated three simple transfer functions to demonstrate the idea. The first one only assigning the same color which is then weighted by the sample colors (from texture mapping) to specify different regions of volume data. The second one applies a commonly used 1D graph to map different domains with different colors, shown as Figure 4.6. The alpha channel is separately assigned. The third one applies a simple linear equation to emphasis the interesting part of the dataset, shown as Equation 4.8. Since the range of gl_FragColor is between 0 and 1, the function actually only keep the scalar value from 0.3 to 0.5. The result of the three transfer functions will be shown in Section 5.

Transfer Function 1: $\text{gl\_FragColor} = \text{SampleColor} \times \text{Color}$ (4.7)

Transfer Function 2:

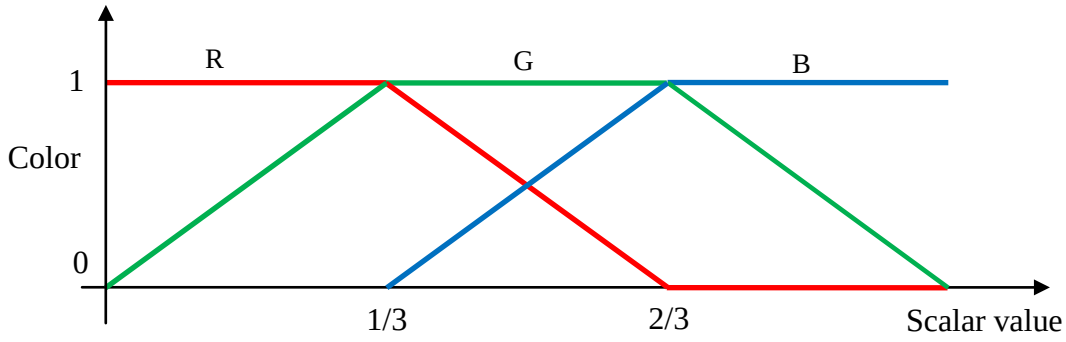


Figure 4.6. Graphic Transfer function

Transfer Function 3:

$$\text{gl\_FragColor} = (\text{SampleColor} - 0.3) \times 2.0 \times \text{Color} \qquad (4.8)$$

The second project used a 1D texture as the transfer function. The texture lookup table is built in the CPU, then load into graphics memory with the function *texture1D(VolumeTexture, TexCoordinate)* . The texture coordinate is the resulting scalar value from *texture3D* function.

4.4 Local Illumination Models

Illumination models are used to improve the visual effect of the rendering objects. Local illumination models only consider light that comes directly from the light sources to the point being shaded. Every point is considered to be separated from all the other pointsb(Fernando, 2004). Traditional local illumination models are built upon the notion of the normal vector, which describes the local orientation of a surface patch and locally approximates the light intensity reflected from the surface of an object.

4.4.1 Blinn-Phong Illumination

The most popular local illumination model in practice is the Blinn-phong model, which computes the reflected intensity as a combination of three illumination phenomenological approximations, ambient, diffuse and specular.

$$I_{phong} = I_{ambient} + I_{diffuse} + I_{specular} \quad (4.9)$$

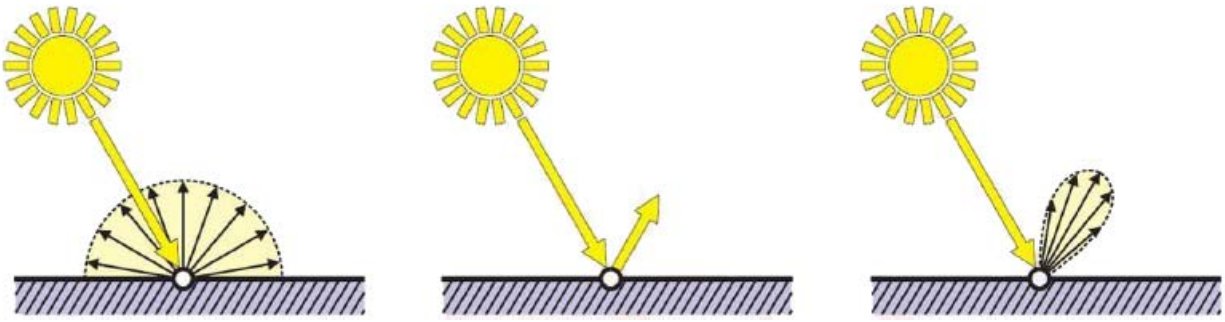


Figure 4.7. The Blinn-phong illumination model (Salama 2006)

The ambient light $I_{ambient}$ is modeled as a constant global light multiplied by the ambient coefficient. It is used to light up the completely black region.

$$I_{ambient} = k_a I_a \quad (4.10)$$

The diffuse part of the model corresponds to the reflection of the surface, which is equally in all directions. Its brightness only depends on the angle between the direction of light *I* and the surface normal *n*. For view-aligned slicing, the direction of the light, is the opposite vector of the view direction, while for ray casting is the opposite vector of the ray direction.

$$I_{diffuse} = k_d \max((I \cdot n), 0) \quad (4.11)$$

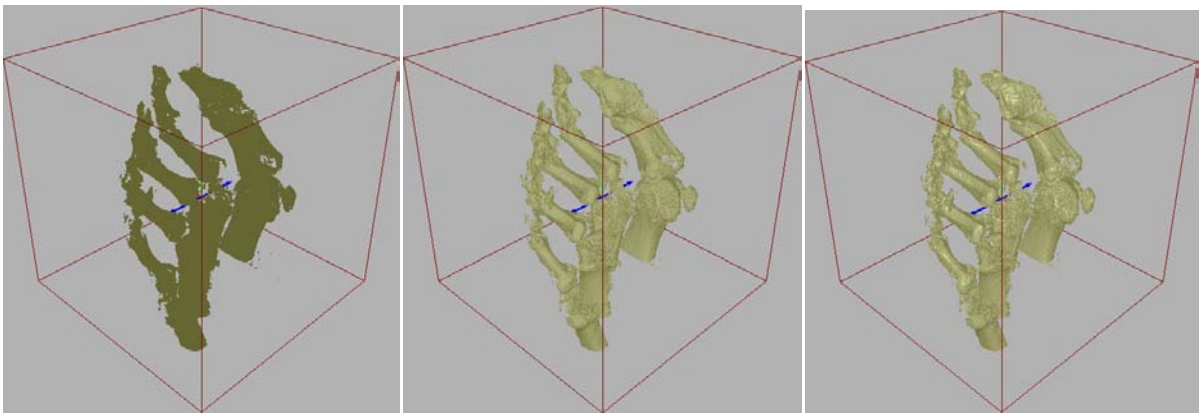
The specular lighting shows the reflection behavior of shiny surfaces, which cause so-called specular highlights. While diffuse is a perfect mirror reflects light in

exactly one direction, the specular light is scattered around the direction of perfect reflection, shown in Figure 4.7. To compute the specular light, vector *h*, which is the halfway between the light direction and the eye direction, has been introduced. For ray casting, because the light direction and the eye direction are all from the camera to the surface point being shaded, so I is used instead of calculating *h*. For view-aligned slicing method, the light position is set to be (1.5, 1.0, 1.0).

$$I_{specular} = k_s \max((h \cdot n), 0)^n \qquad (4.12)$$

In which the specular exponent n is called shininess of the surface, which controls the size of the resulting highlights.

Figure 4.8 (a) shows the foot with the ambient light, (b) adds the diffuse light and (c) displays the final rendering outcome of the illumination model including ambient, diffuse and specular light.



(a) *(b)* *(c)*

Figure 4.8. The result Blinn-phong illumination mode

4.4.2 Gradient-Based Illumination

In order to use Blinn-phong illumination model to discrete volume data, the external light is assumed to be reflected at iso-surfaces, which is the surface that results from tracing a specific field value within a dataset, inside the volume data. The normal used for shading a point is thus the unit vector which is perpendicular to the iso-surface through that point. Considering that the gradient vector of the scalar field points into the direction of greatest change, which is always perpendicular to the surface, we estimate the gradient vector to approximate the normal vector for local illumination. (Fernando, 2004)

The gradient vector is the first-order derivative of the scalar field, as Equation 4.13. The normalized gradient is used as the normal, and the gradient magnitude is a scalar quantity which describes the local rate of change in the scalar field (Engel et al 2006).

$$\nabla f(\vec{x}) = \left(\frac{\partial f(\vec{x})}{\partial x} \quad \frac{\partial f(\vec{x})}{\partial y} \quad \frac{\partial f(\vec{x})}{\partial z}\right)^{T} \qquad (4.13)$$

4.4.3 Gradient Estimation

There are various techniques to calculate the gradient from discrete volume data. In our project the gradient estimation is computed in real-time on a per-pixel basis in the fragment shader. There are variety of methods for estimate the directional derivatives, such as finite differences and convolution filtering for gradient

estimation. They may have different complexity and accuracy according to the different methods. Finite differencing scheme, as a fast and efficient method for estimating gradients from discrete volume data, are used in our project.

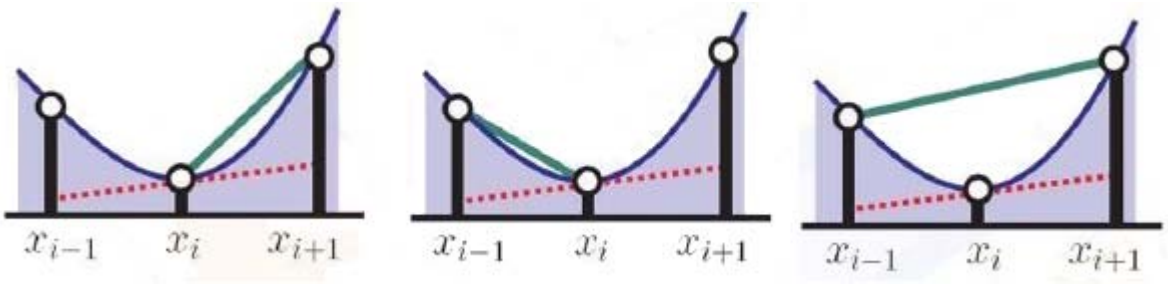
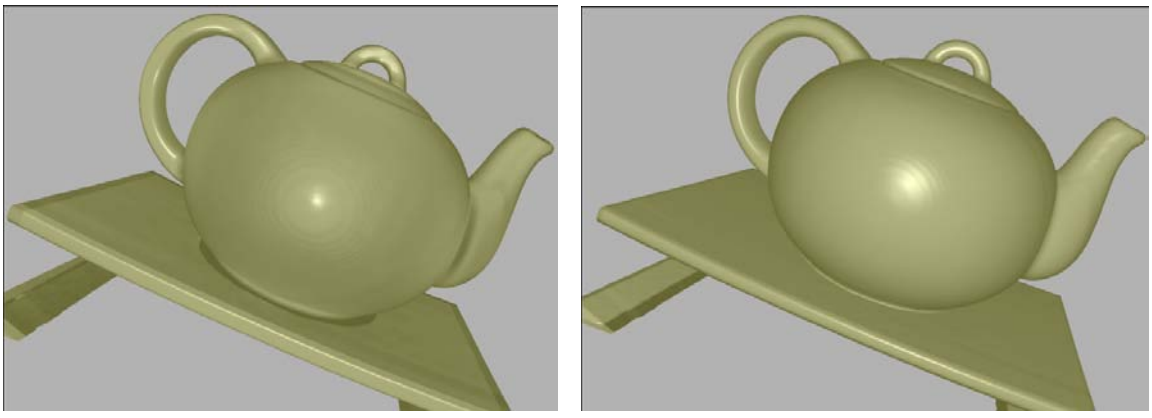


Figure 4.9. Finite differencing schemes (Engel et al 2006)

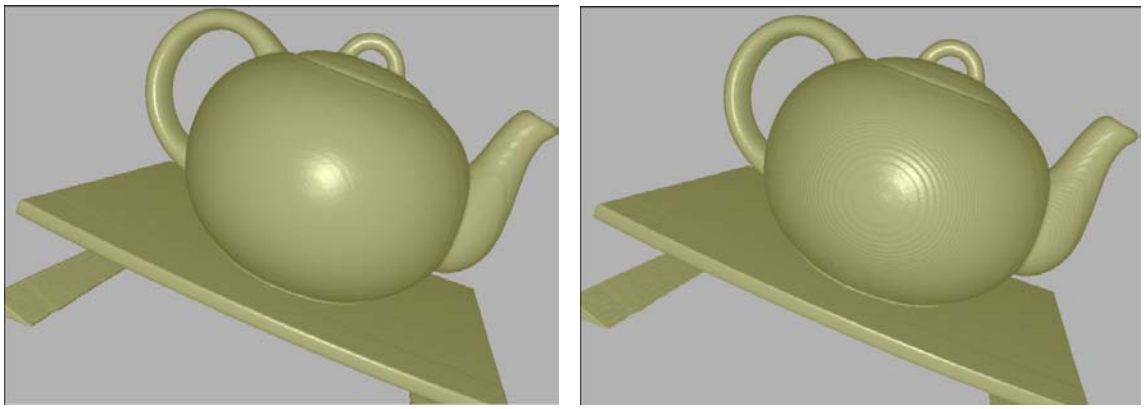
The result of the first-order derivative of a 1D scalar function f(x) in the point x_i is called central differences with an approximation error (Engel et al 2006):

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h} + o(h^2) \qquad (4.14)$$

In which, $x_{i+1} = x_i + h$, $x_{i-1} = x_i - h$. *h* is the step size. The smaller *h* is, the smaller the approximation error is. However, if h is too small, we may get artifacts, shown as in Figure 4.10. The step size are separately 0.025, 0.01, 0.0025 and 0.0005. It is obvious that 0.025 is too big to get a proper output, while 0.0005 is too small and result in visual artifaces.



(a) step size= 0.025 *(b) step size= 0.01*



(c) step size= 0.0025 *(d) step size= 0.0005*

Figure 4.10. Illumination mode with different step size of gradient calculation

Each of the three components of the gradient vector $\nabla f(\vec{x}) = \nabla f(x, y, z)$ is proximate by a central difference, resulting in (Engel et al 2006)

$$\nabla f(x, y, z) \approx \frac{1}{2h}\begin{pmatrix} f(x+h, y, z) - f(x-h, y, z) \\ f(x, y+h, z) - f(x, y-h, z) \\ f(x, y, z+h) - f(x, y, z-h) \end{pmatrix} \qquad (4.15)$$

4.5 Compositing

Compositing is fundamental for the iterative computation of the discrete volume rendering integral. It defines how the color values of the textured polygons that we draw are successively combined to create the final rendition. The compositing equation depends on the traversal order. The front-to-back iteration equations are used when the viewing rays are traced from the eye position (camera) into the volume. The back-to-front compositing scheme is used when the data set is traversed from its backside.

Back-to-front compositing (Fernando 2004):

$$\hat{C}_i = C_i + (1 - A_i)\,\hat{C}_{i+1} \qquad (4.16)$$

$$\hat{A}_i = A_i + （1 - A_i）\hat{A}_{i+1}$$

Where C_i and A_i are the color and opacity obtained from the fragment shading stage for sample i along the viewing ray, and $\hat{C}_i$ and $\hat{A}_i$ is the accumulated color and opacity from the back of the volume.

Front-to-back compositing:

$$\hat{C}_i = (1 - \hat{A}_{i-1})C_i + \hat{C}_{i-1} \qquad (4.17)$$

$$\hat{A}_i = (1 - \hat{A}_{i-1})A_i + \hat{A}_{i-1}$$

Where $\hat{C}_i$ and $\hat{A}_i$ is the accumulated color and opacity from the front of the volume.

The compositing equations specify a combination of the RGBA quadruplet of an incoming fragment (*source*) with the values already contained in the frame buffer (*destination*). They are easily implemented with hardware ***alpha blending***. If the blending is disabled, the destination value will be replaced by the source value. While the blending is enabled, the source and the destination RGBA quadruplet are combined by a weighted sum forming a new destination value.

For the front-to-back compositing, the source blending factor is set to 1 and the destination blending factor is set to (1- source alpha). It is important to note that this blending set-up uses associated colors (Blinn 1994), which are already weighted by their corresponding opacity. Therefore, OpenGL applications often use a different equation for back-to-front blending, denoted

$$\hat{C}_i = A_i \cdot C_i + （1 - A_i）\hat{C}_{i+1}$$

Therefore, the stand alpha blending setup is as followed

```
glEnable (GL_BLEND);
glAlphaFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
```

Which is used in this project.

5. Result and Analysis

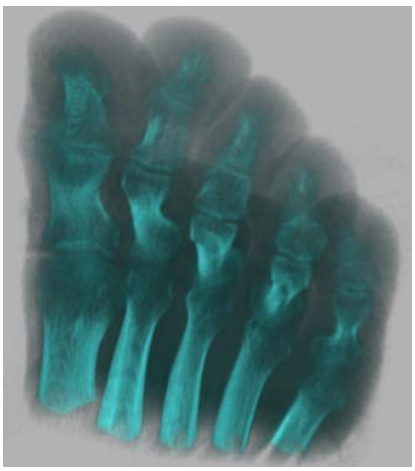
The rendering result strongly depends on the structure of the data set, the chosen transfer function, the current view direction, and the sampling rate. This section will look at examples of some of the results achieved from the project, along with an analysis on certain features.

5.1 Object-Aligned Slicing method

Transfer function

The three transfer functions mentioned in Section 4.3 were implemented by six volume data sets. From the rendering outputs shown as bellowed, It can be concluded that one transfer function may work well for certain volume data, but may not suitable for the others. Therefore, transfer function needs to be modified when applying to different data sets. The distance between adjacent slices for this test is set to be 0.01, the wooden pattern artifacts can be seen clearly from some of images.

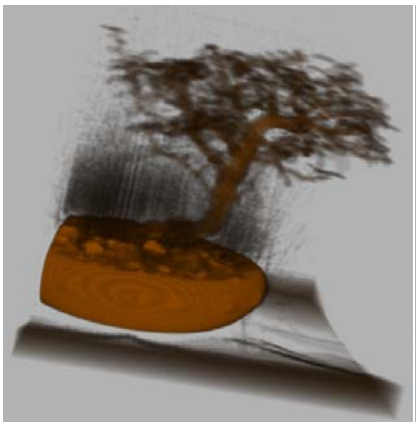
a) Foot

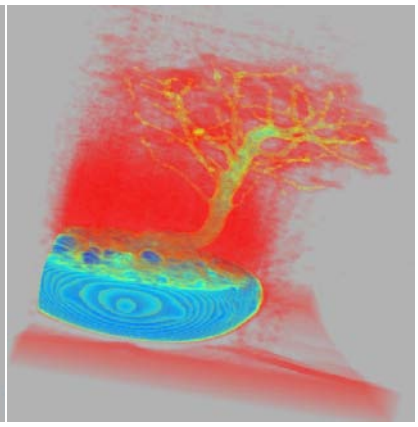

Function 1

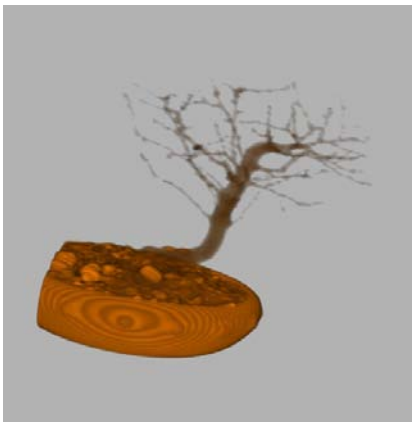

Function 2


Function 3

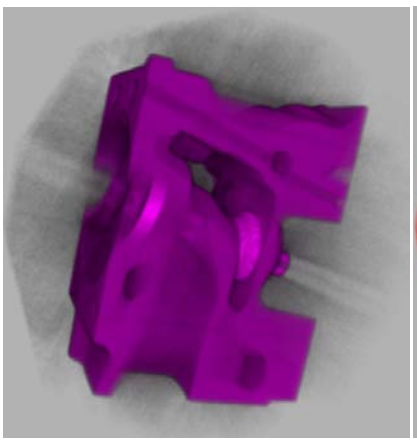
b) tree

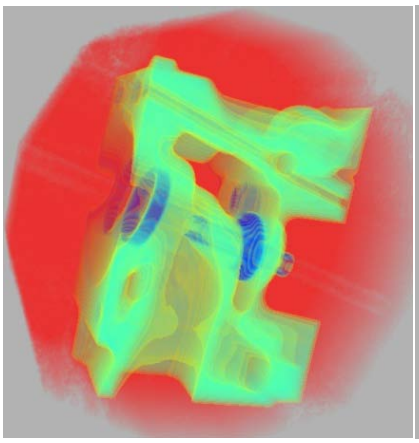

Function 1

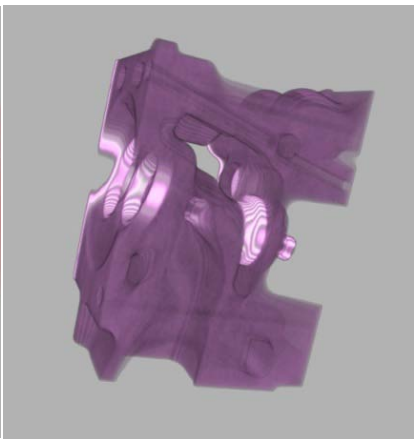

Function 2


Function 3

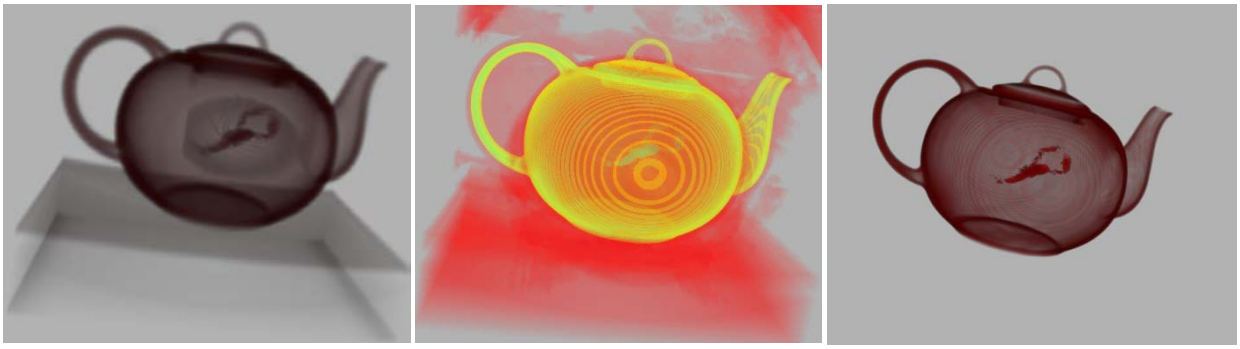
c) engine


Function 1


Function 2

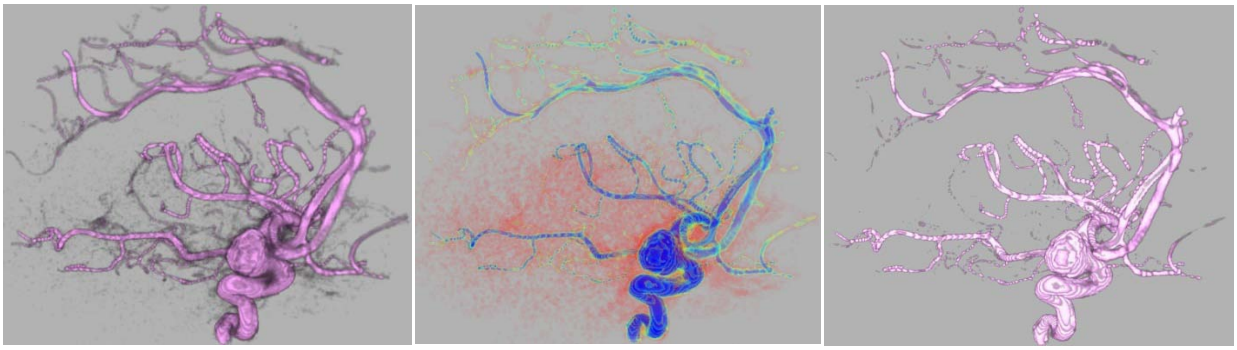

Function 3

d) teapot



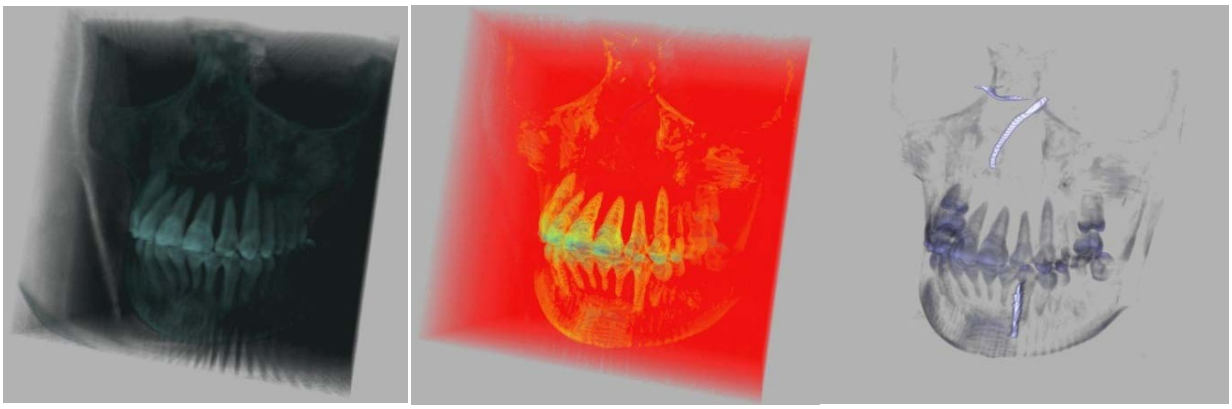
Function 1 Function 2 Function 3

e) aneurism



Function 1 Function 2 Function 3

f) skull



Function 1 Function 2 Function 3

Figure 5.1. Object-aligned slicing rendering results

Texture Slice

Textured slices can let the users observe more details inside the volume data. The

major axis can be chosen from x, y, z axises and diagonal direction to fit the different types of the volume data.

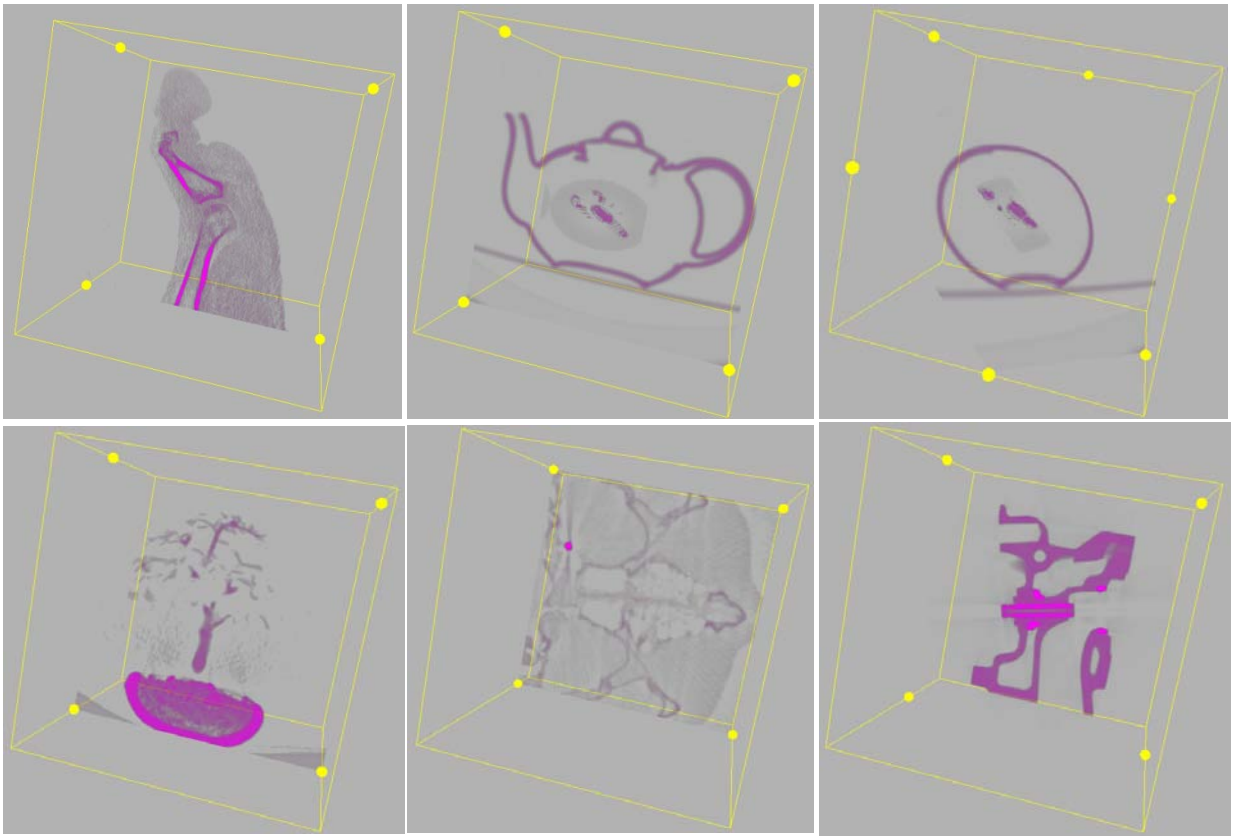


Figure 5.2. Object-aligned slicing rendering, showing single slice

5.2 View-Aligned Slicing method

The view-aligned slicing method adopted the Blinn-Phong illumination models.

Illumination Shader

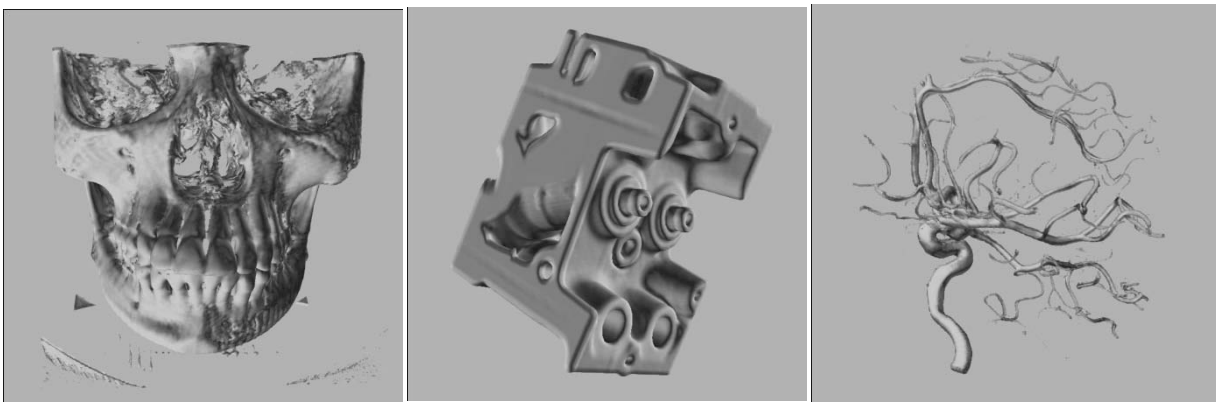
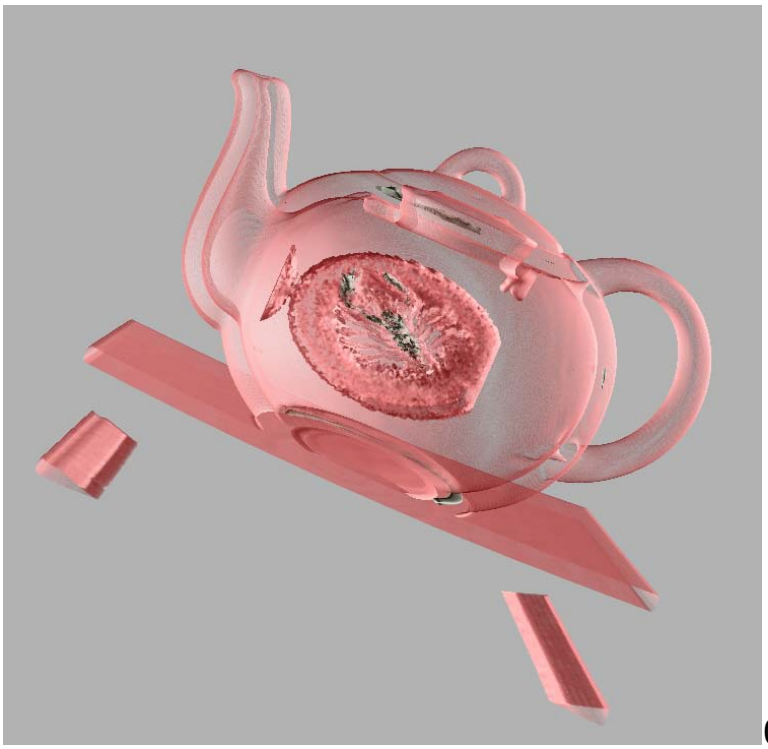


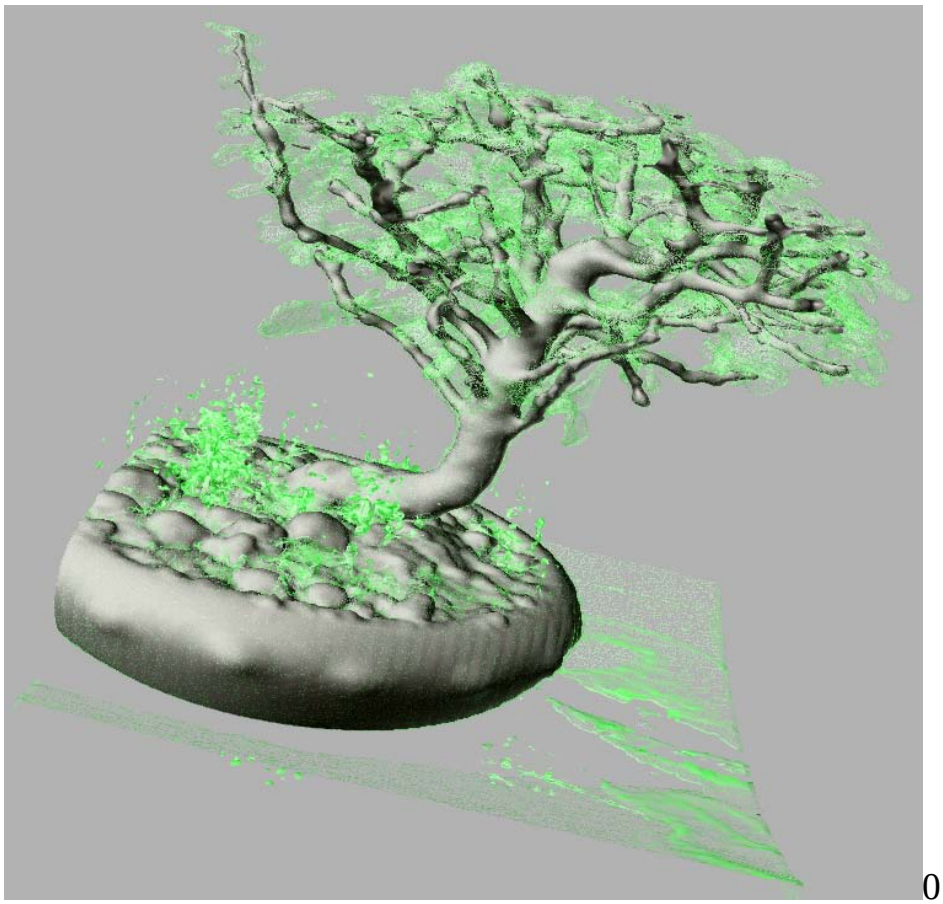
Figure 5.3. Illumination model

Multi-layer Shader

Iso-surface rendering achieves a good result in emphasizing the important features of volume date, however, it suffers from the drawback of without preserving the most information of the volume data. To overcome the single iso-surface's drawback, a combination of iso-surfaces and volume rendering test has been implemented in this project to enrich the visualization effects. Different features were extracted according to the varied scalar values sampled from the 3D texture.



0.04-0.055



0.124-0.129

0.048-0.050

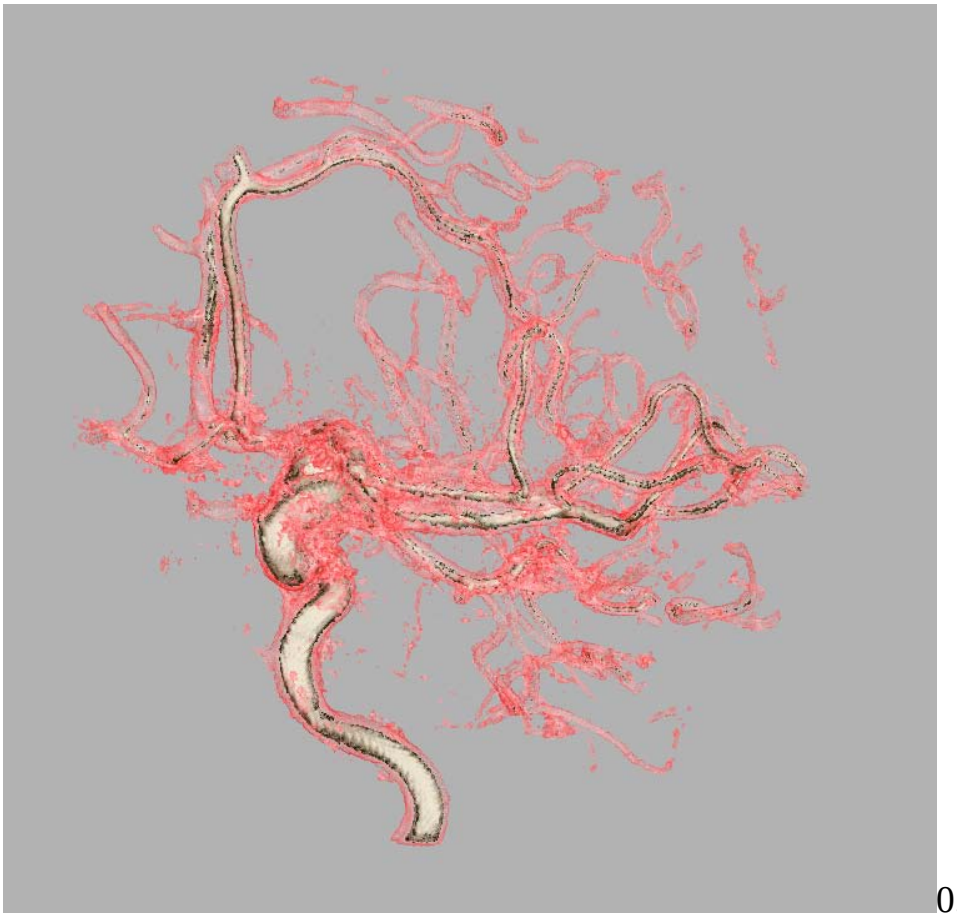


Figure 5.4. multi-layer shader for view aligned slicing texture

The numbers besides the image are the color-sample ranges of the outside layers, the iso-value of the inside layer can be controlled by the user from the interface.

1D Color-table Look-up Transfer Function

1D color table look up texture can simultaneously assign different colors to different parts of one volume object, which can distinguish different features of the volume data. The user needs to change the color array to correspondingly change the 1D texture transfer function.

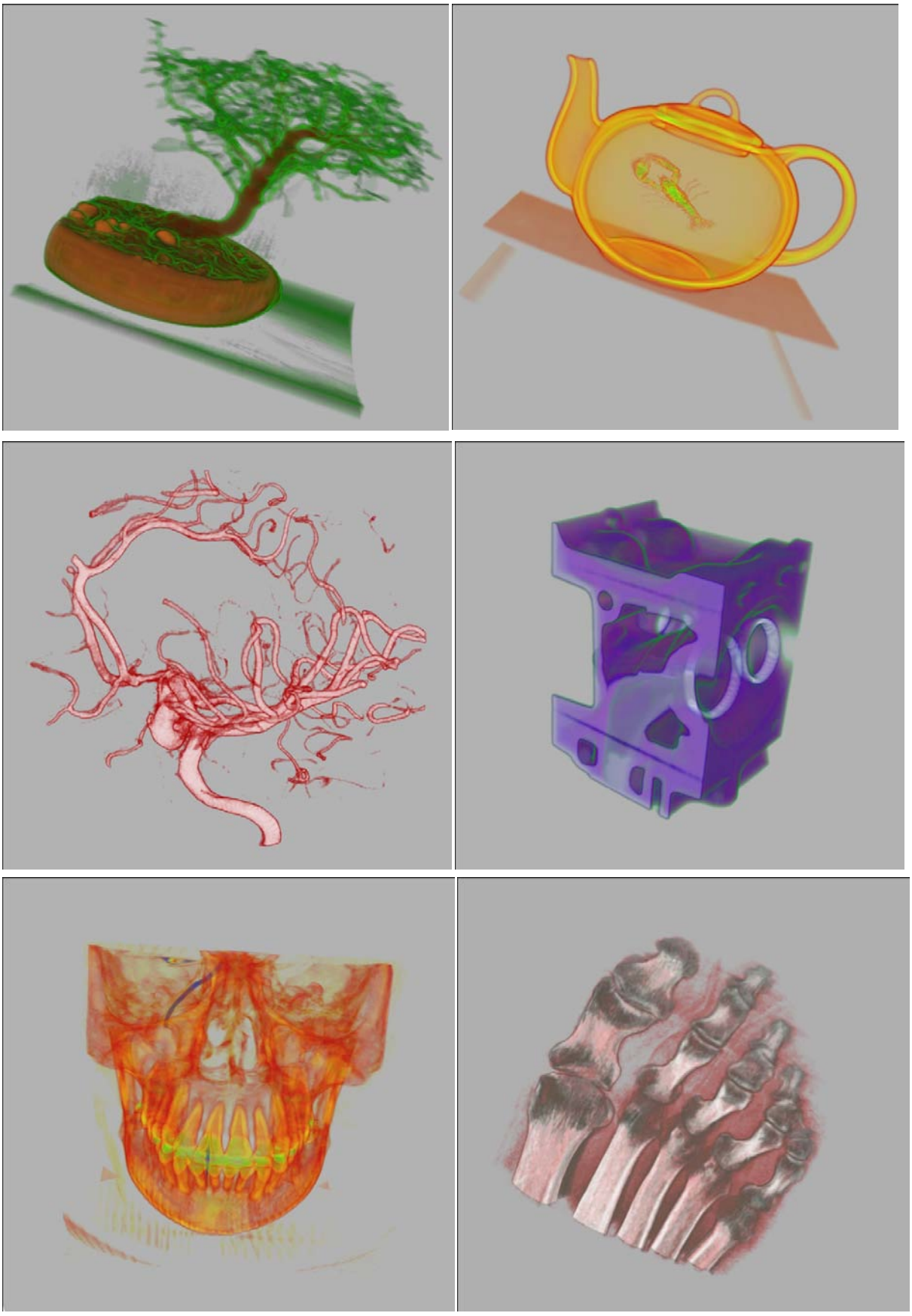


Figure 5.5. 1D Transfer Function Texture of texture slicing method

5.3 Ray casting

Illumination Shader

Illumination shader used for ray casting is quite similar with the view-aligned slicing method. A series of foot images shown below illustrate the different iso-surfaces with the different scalar value sampled from the 3D texture.

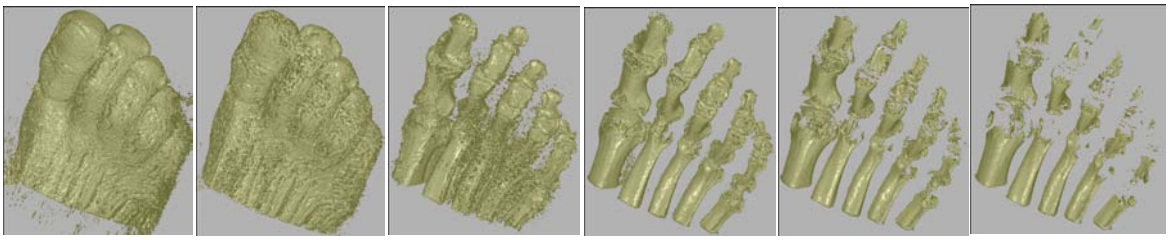
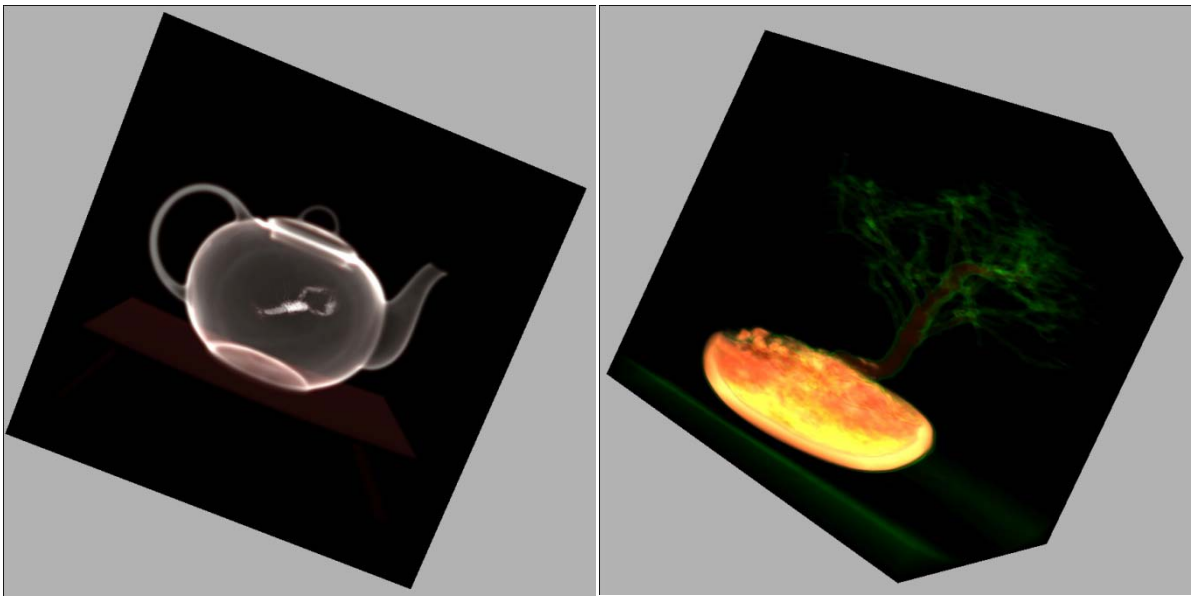


Figure 5.6. Illumination model animation

1D Color-table Look-up Transfer Function

The theory of color-table look-up transfer function for ray casting is the same with the view-aligned approach.



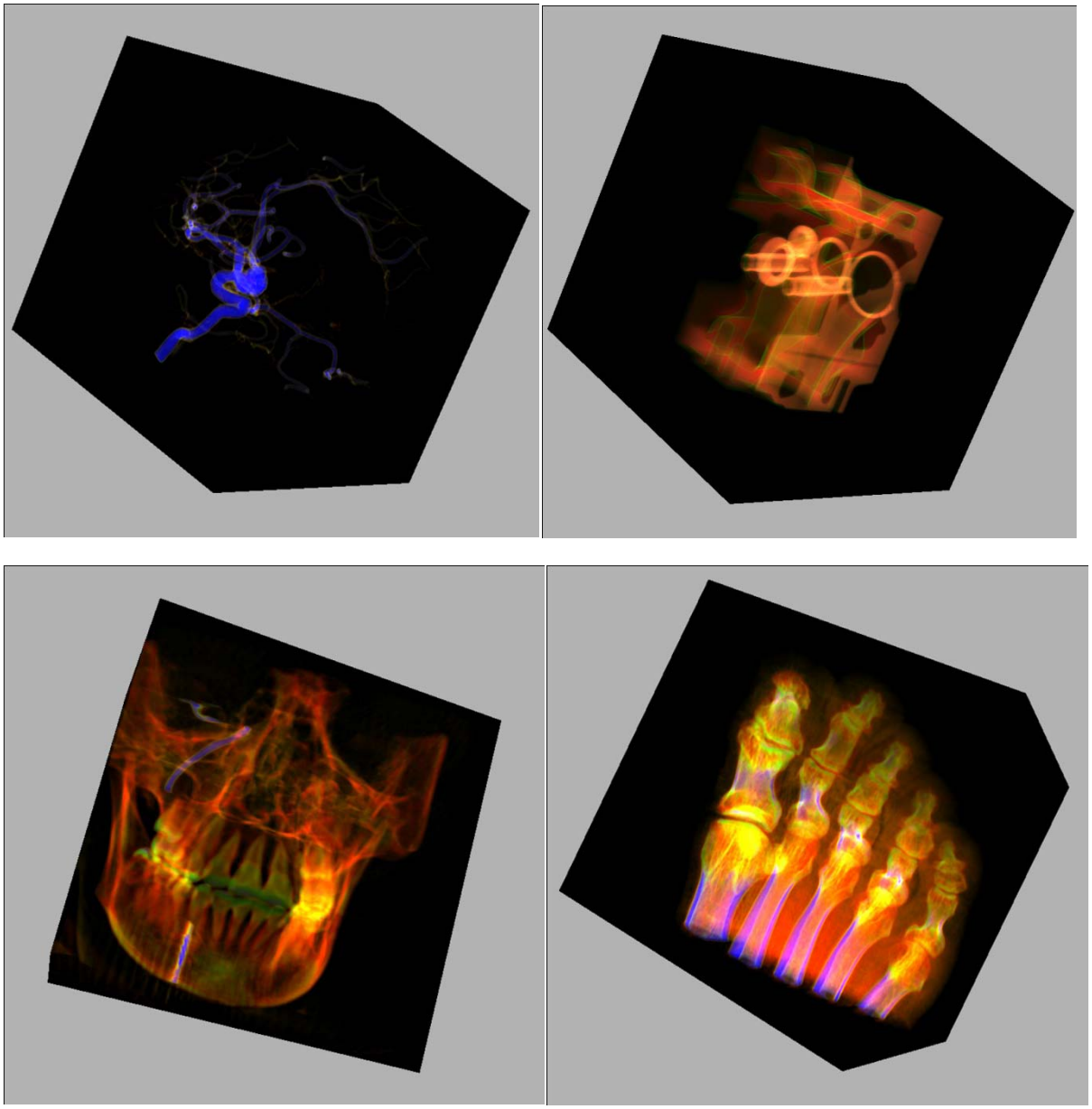
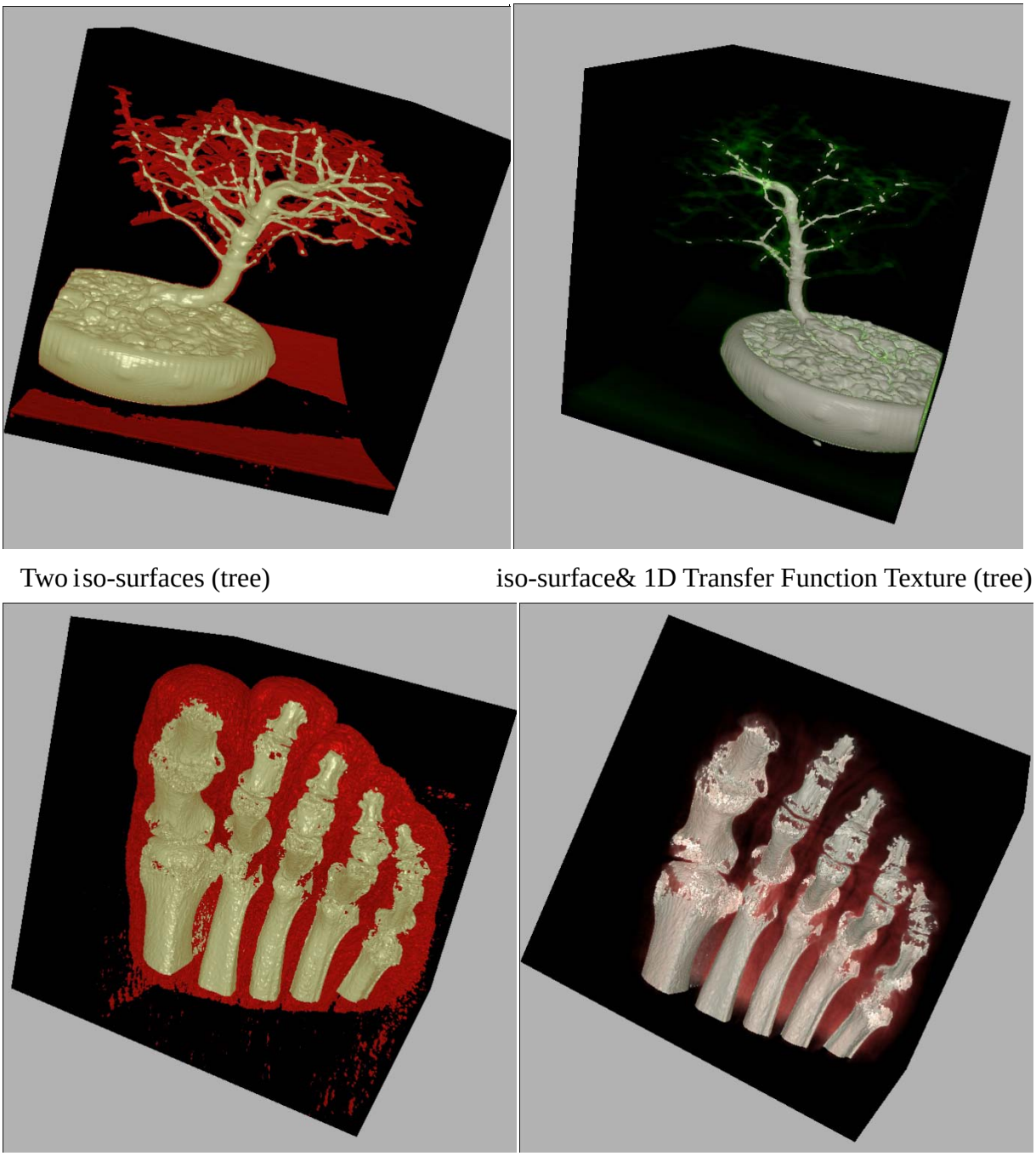


Figure 5.7. 1D Transfer Function Texture of ray casting

Multi-layer Shader

For ray casting, two kinds of multi-layer shaders have been implemented. One is the combination of two iso-surfaces, the other one is the combination of the iso-surface with 1D Color-table Look-up Transfer Function.



Two iso-surfaces (tree) iso-surface& 1D Transfer Function Texture (tree)

Two iso-surfaces (foot) iso-surface& 1D Transfer Function Texture(foot)

Figure 5.8. multi-layer shader for ray casting

5.4 Efficiency Comparison

The efficiency comparison between GPU-based ray casting and view-aligned texture slicing methods are measured by the frame rates, see Table 5.1.

	Texture Slicing		Ray Casting	
Step Size	0.01	0.001	0.01	0.001
Foot	36	5	131	23
Teapot	44	5	132	23
Aneurism	45	5	116	17
Tree	37	5	120	20
Skull	48	5	158	24
engine	41	5	146	21

Table 5.1. Comparison between texture slicing and ray casting volume rendering approaches. The figures represent average frame rates.

With the same sampling rate for ray casting and texture slicing methods, the single pass ray casting runs much faster than texture slicing. The reason why ray casting is superior to texture slicing is mainly because ray casting only renders a single polygon to generate the necessary fragments, while texture slicing deals with thousands of polygonal slices. Ray casting performs a very low geometry processing and fragment generation overhead.

6. Conclusion

In this project, I used volume rendering technique which doesn’t require any specialized hardware to achieve real-time 3D modeling. Local illumination models and transfer function have been used to improve the quality of rendering and creating volumetric special effects. With the limited time to implement features, there are some of the applications can be improved and a lot of work can be fit into the current framework with future work.

1) The efficiency of GPU-based volume rendering algorithm is always limited by the ability of fragment processor because the major workload is handled by fragment processor. When the sampling distance decrease, the number of fragment needs to be processed will increase sharply, which will result in a very slow interactive operation. With even more larger data sets, the higher sampling rate and image resolution requires us to find way to access the volume data in an more efficient way. One solution is by “performing ‘expensive’ computations and accessing memory only selectively” and applied some advanced methods like “leaping over empty space, skipping occluded parts, and termination of rays” to help achieve the goal. (Engel et al 2006)

2) From this project, we can see that the 1D Color-table Look-up texture transfer function can’t achieve a high resolution image. Moreover, 1D transfer function can’t deal with data value with multiple boundaries. Therefore, there is now a trend toward using multidimensional transfer function, which can capture the relationship between multiple data values and create more stunning visualization effects. (Kniss Kindlmann Hansen 2002) Curvature-based transfer function as a multidimensional transfer function has received much attention recently. (Kindlmann et al 2003) There is also a trend towards using gradient magnitude in the transfer function domain, since it can be used to emphasis the domain where has the biggest changes, thus the material boundaries can perform a strong visual effects.

7. Bibliography

Akeley, K., 1993. *RealityEngine Graphics*. SIGGRAPH ’93 Proceedings of the 20th annual conference on Computer Graphics and interactive techniques, NY: ACM.

Blinn. J. F., 1994. *Jim Blinn’s Corner: Image Compositing-Theory.* IEEE Computer Graphics and Applications. 14(5) 83-87.

Csebfalvi, B., Konig, A. and Groller, E., 2000. *Fast Surface Rendering of Volume Data*. International Conference in Central Europe on Computer Graphics and Visualization - WSCG ,

Cullip, T. J., Neumann, U., 1993. *Accelerating Volume Reconstruction With 3D Texture Hardware*. Technical Report. NC: University of North Carolina at Chapel Hill.

Engel, K., Hadwiger, M., Kniss, J. M., Rezk-Salama, C., 2006. *Real-time volume graphics.* MA: A K Peters, Ltd.

Fernando, R., 2004. *GPU Gems*. US: NVIDIA Corporation.

Fernandes, R. A., 2011. GLSL Tutorial. Available from: http://zach.in.tu-clausthal.de/teaching/cg_literatur/glsl_tutorial/index.html [Accessed 4 August 2011]

Jin, J., Wang, Q., Shen, Y. and Hao J., 2006. *An Improved Marching Cubes Method for Surface Reconstruction of Volume Data.* Intelligent Control and Automation. Dalian: WCICA

Kajiya, J. T., 1984. *Ray Tracing Volume Densities.* ACM SIGGRAPH Computer Graphics. NY: ACM. 18(3) 165-174.

Kindlmann, G., Whitaker, R., Tasdizen, T. and Moller, T., 2003. *Curvature-based transfer functions for direct volume rendering: methods and applications.* IEEE Transactions on Ultrasonics Ferroelectrics and Frequency Control (2003), 513-520.

Kniss, J., Kindlmann, G. and Hansen, C., 2002. *Multidimensional Transfer Functions for Interactive Volume Rendering.* IEEE Transactions on Visualization and Computer Graphics, 8(3) 270-285.

Kruger J. and Westermann R., 2003. *Acceleration Techniques for GPU-based Volume Rendering*. In Proceedings of IEEE Visualization '03. 2(3) 287-292.

Lee, H., Desbrun, M., Schroder, P., 2003. *Progressive Encoding of Complex Isosurfaces.* NY: ACM. ACM Transactions on Graphics (TOG)-Proceedings of ACM SIGGRAPH 2003, 22(3).

Levoy, M., 1988. *Display of Surfaces from Volume Data.* IEEE Computer Graphics Applications. 8(2) 29-37.

Li, W., Mueller, K. and Kaufman, A., 2003. *Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering*. In Proceedings of IEEE Visualization'03, 317-324.

Lorensen, W. E., and Cline, H. E., 1987. *Marching cubes: A high resolution 3D surface construction algorithm.* Computer Graphics, 21(4) 163-169.

Moreland, D. K., 2004. *The Volume Rendering Integral*. NM: University of New Mexico Department of Computer Science. Available from: http://www.cs.unm.edu/~kmorel/documents/dissertation/thesis_full/node8.html [Accessed 4 August 2011].

Pawasauskas, J., 1997. *Volume Visualization With Ray Casting.* MA: Worcester Polytechnic Institute. Available from: http://web.cs.wpi.edu/~matt/courses/cs563/talks/powwie/p1/ray-cast.htm

[Accessed 4 August 2011].

Rottger, S., Guthe, S., Weiskopf, D. and Ertl, T., 2003. *Smart Hardware-Accelerated Volume Rendering*. VISSYM's 03 Procceedings of the Symposium on Data Visualization. 2(3) 231-238.

Salama, R. C., 2006. Real-Time Volume Graphics Tutorial. Available from: http://www.real-time-volume-graphics.org/?page_id=28
[Accessed 4 August 2011].

Shirley, P. and Tuchman, A., 1990. *A polygonal Approximation to Direct Scalar Volume Rendering*. Computer Graphics. 24(5) 63-70.

Stegmaier, S., Strengert, M., Klein, T. and Ertl, T., 2005. *A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting.* Volume Graphics 2005, 187-241.

Sunday, D., 2001. *Intersections of Rays, Segments, Planes and Triangles in 3D.* Available from:
http://softsurfer.com/Archive/algorithm_0105/algorithm_0105.htm#Segment-Plane . [Accessed 4 August 2011].

University of Tübingen, 2005. *Volume Dataset Repository*. Available from: http://www.volvis.org/ [Accessed 4 August 2011].

Weiler, M., Kraus, M., Merz, M., Ertl, T. 2003. *Hardware-Based Ray Casting for Tetrahedral Meshes*. In Procceedings of 14th IEEE Visualization 2003, 2(3) 333-340.

WestOver, L., 1990. *Footprint Evaluation for Volume Rendering.* Procc. SIGGRAPH' 90, Computer Graphics. 24(4) 367-376.