GPU-Friendly Marching Cubes

XIE, Yongming

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
Computer Science and Engineering

Abstract of thesis entitled:
GPU-Friendly Marching Cubes
Submitted by XIE, Yongming
for the degree of Master of Philosophy
at The Chinese University of Hong Kong in February 2008

Marching cubes has long been employed as a standard indirect volume rendering approach to extract isosurfaces from 3D volumetric data. This thesis presents a GPU-friendly MC implementation. Besides the cell indexing, we propose to calculate vertex and normal interpolations by precomputing the expensive equations and looking up these values during runtime. Upon a commodity Graphics processing unit (GPU), our implementation can rapidly extract isosurfaces from a high-resolution volume and render the result. With the proposed parallel marching cubes algorithm, we can naturally generate layer-structured triangles, which facilitate the visibility-correct visualization of multiple-layer translucent isosurfaces without performing computational expensive sorting. The algorithm extracts and draws triangles, in a layer by layer fashion, from back to front. With the painters algorithm, the visibility of multi-layer translucent isosurfaces is resolved naturally.

論文摘要

Marching cubes (MC) 作爲一種間接的標準體繪製途徑，長期以來被用于三維的體數據表面抽取。 在這篇論文中，我們報告了一種利用圖像硬件加速 MC 表面抽取與繪製。首先我們對費時的計算預先處理，在 MC 單元索引的時，根據預先處理結果對所需要的點位置與法綫進行綫性的插值進行查值計算。 在一個通用的圖像處理上，我們提升了從高分辨率的體數據中表面抽取的速度與渲染的結果。 在我們報告的方法中，我們可以隨意的獲取層狀結構的三角形，不需要對這些三角形進行排序就可以得到一個正確的透明的結果，同時還可以進行多層次的透明繪製。這個方法采用了從後面到前面一層一層的繪製，這樣就可以得到正確的透明繪製結果， 通過這樣繪畫的方法，多層次的表面透明可以得到自然的解決。

Acknowledgement

First, I would like to thank my supervisor Dr. HENG Pheng Ann , who has patiently guided me through three years of my M.Phil. study. Without his encouragement and guidance I could not finish my research. I would also like to thank my thesis committee, Dr. WONG Tien Tsin, Dr. LEUNG Kwong Sak and Dr. WU En Hua. Dr. LEUNG Kwong Sak and Dr.WONG Tien Tsin have been my markers many times and given me many useful advices on my research work. I want to thank my colleague,WANG Guangyu , who always gave me valuable advise and encourage when I felt frustrated with my research. I would like to thank my dear friends in CUHK. The life with you will be part of my memory. Finally, I would like to express my deepest gratitude to my family, your love is the most importance for me. The work described in the thesis was substantially supported by a grant from the Research Grant Council of the Hongkong Special Administrative Region, China. (Project No.CUHK 4461/05M)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Our research aims at proposing algorithm that can rapidly extract and render isosurfaces from high-resolution 3D volume data as well as correctly visualize multiple layers of translucent isosurfaces, without sorting

1.1 Isosurfaces

Isosurfaces are normally displayed by computer graphics, and are usually used as data visualization methods to allow us to study features of a volume object. Isosurfaces tend to be a popular form of visualization for volume datasets since they can be rendered by a simple triangular model, which can be drawn on the screen very quickly. In medical imaging, isosurfaces may be used to represent regions of a particular density in a three dimensional CT scan, allowing the visualization of internal organs, tissue, or other structures. Isosurfaces also have been widely adopted to reveal the complex structures in medical and scientific volume data because of its fine visual quality. Visualizing multiple layers of translucent isosurfaces (normally represented as triangles) not just generates high-quality rendering results, but also allows viewers to better understand the relationship among internal structures . However, visibility-correct visualization of multiple translucent isosurfaces imposes a lot of difficulties. For example, standard depth-buffering alone cannot resolve the visibility of overlapped translucent triangles. On the other hand, extracted triangles can be sorted by depth sorting or binary space partitioning (BSP) based visibility sorting in order to generate a correct drawing order. In other words, computational expensive sorting has to be performed whenever the viewpoint changes. In this thesis, we propose a GPU-friendly isosurface extraction method that facilitates the visibility-correct visualization of multiple layers of translucent isosurfaces. Instead of performing visibility sorting, triangles are drawn from the back to front in a layer-by-layer fashion, i.e. the painter's algorithm.

1.2 Graphics Processing Unit

As technology advances, graphics cards become fully programmable, which support rendering and computing. With the rapid progress in Graphics processing unit (GPU), various applications associated with computer graphics advance greatly. At the same time, the processing power, parallelism and programmability available on the current GPU provide an ideal platform for general-purpose computation such as algebraic computation, Nowadays, while a 3.0 GHz Pentium IV can perform 6 Gflops in theory, a GeForce 7800 GPU by NVIDIA gives a performance up to 313 Gflops. In this thesis, we will demonstrate that GPU can do more than just rendering. our approach will exploits OpenGL framebuffer object [62] to store the rendered texture and/or transfer it from GPU framebuffer to CPU memory. The parallelism makes it be a fast platform for many computer graphic problems as well as other general computational questions. However, there are limitations on GPU for its stream programming model. This motivates us to rethink how we solve certain problems. As GPUs continue to grow at a rapid pace, it is likely that GPU is becoming a mainstream for general-purpose computation.

1.3 Objective

The objective of this thesis is to propose a GPU-friendly MC algorithm to speedup the extraction and rendering of isosurfaces. The proposed algorithm can rapidly extract and render isosurfaces from high-resolution volume data. In addition, this algorithm can correctly visualize multiple layers of translucent isosurfaces, without sorting.

1.4 Contribution

In this thesis, we proposed a algorithm that can rapidly extract and render isosurfaces from high-resolution 3D volume data. Our framework can be trivially modified to implement a wide range of MC variants. With this framework, we can correctly visualize multiple layers of translucent isosurfaces, without sorting. The proposed framework ensures the extracted triangles are drawn in a correct order, from back to front, according to the viewing direction. The extracted geometry is stored in GPU memory and they can be transferred to main memory for further processing. Given the geometry, many other interesting applications can be developed. In addition, this approach allows us to visualize the complex translucent isosurfaces in real time. The layer-structured triangles are directly generated by the proposed GPU-based isosurface extractor according to the user-specified isovalues.

1.5 Thesis Organization

The rest of this thesis is organized as follows. In Chapter 2, the theory and general framework of MC algorithm are described. Chapter 3 depicts the history of GPUs, the processing pipeline of GPUs, and their limitations. In Chapter 4, we give a detailed introduction of existing volume rendering techniques. In Chapter 5,we present the general framework to allow MC algorithm to be executed on GPU. In addition, an advanced algorithm for GPU implementation of visualizing multiple layers of translucent isosurfaces is described in detail. Finally, conclusions are given in Chapter 6.

□ **End of chapter.**

Chapter 2

Marching Cubes

2.1 Introduction

The Marching Cubes algorithm [44] is a famous technique for extracting isosurface from 3D volumetric data. It was originally developed by Lorensen and Cline in 1987. Before applying the Marching Cubes algorithm (MC)to extract and reconstruct 3D surface, the volume data should to be partitioned into cubes. Before introducing the algorithm, we list some basic conceptions in Figure 2.1.

Cube	The volume defined by eight neighboring points
Vertex	The pixel values at the eight corner points of the cube
Face	One of the six sides of a cube
Edge	One of the four rims of a face
Isosurface	All points within the cube with equal property
Isovalue	The value of the material property

Figure 2.1: Some basic conceptions of Marching Cubes algorithm.

The basic principle is as follows. We can define a cube by eight voxels volume elements at the corner of cube and subdivide the whole volume into a series of small cubes [69]. If one or more voxels of a cube have values less than the isovalue, which is user-specified and represents the interesting material property. And one or more have values greater than this value; the voxel must contribute some components of the isosurface. By determining which edges of the cube are intersected by the isosurface, a surface of up to four triangles is placed inside the cube. Then the algorithm "marches" on to the small cube in next scan line order. The Marching Cubes algorithm identifies 256 configurations for the cube, depending on whether the eight vertices are inside or outside the object. As shown Figure 2.2, the blue points are inside the object, and red points are outside the object. Three triangles are inserted into the current cube to separate the blue ones and the red ones.

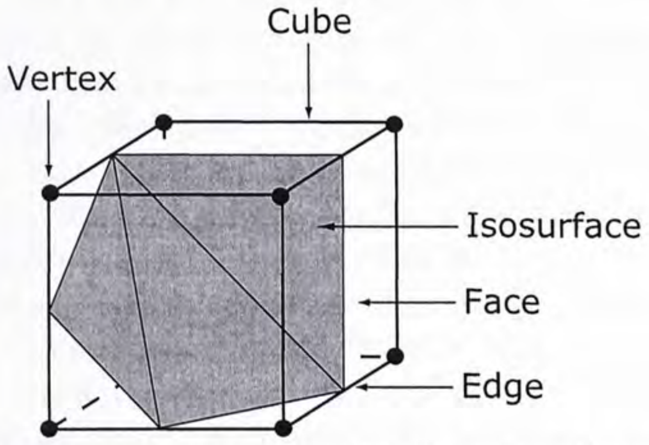


Figure 2.2: Each vertex is inside or outside the surface. Vertex is 0 when it is outside the surface; and 1 when it is inside the surface..

Marching Cubes algorithm uses linear interpolation between voxel values to compute the location of the triangle's vertices. The result of all cubes in this way is a collection of surface, which approximate the shape of the isosurface. Based on its original conception, the Marching Cubes algorithm has been the subject of much further research to improve the quality of its surface representation and performance on large data sets. The advantage of the Marching Cubes algorithm is that the resulting triangle model can be displayed by standard rendering algorithms on the traditional graphics card. It uses information from the original 3D volume data to derive inter-voxel connectivity, surface location, and surface gradient. In addition, because the algorithm uses a case table of the edge intersections to describe how a surface cuts through each cube, the time performance of this algorithm is well, which is an important factor for realtime applications.

2.2 Marching Cubes Algorithm

Volumetric datasets are generally organized as 3D rectilinear grids with a scalar value stored at each grid point. The algorithm uses a divide-and-conquer approach to locate the surface in a logical cube created from eight pixels [44]; four each from two adjacent slices as shown in Figure 2.3. The algorithm marches each of the cubes in the volumetric datasets. The algorithm determines how the surface intersects this cube, then marches to the next cube. The pixel is assigned to a cube's vertex if the data value at that vertex exceeds (or equals) the value of the surface are constructed. These vertices are in or on the surface. Cube vertices with values below the surface and are outside the surface. The surface intersects those cube edges where one vertex is outside the surface. To test the corner points, and to replace the cube with appropriate set of polygons, the set of polygons of this algorithm are decided how to define the edge configuration and triangle configuration.

Vertex index of 0-7 vertices and Edge index of 0-11 edges of each cube are indexed as shown in Figure 2.4.

The algorithm proceeds as follows:

Inputs are the threshold value and structured volumetric data set. Output is a triangle mesh which is an approximation of iso-surface. Main steps: Cell construction from given volume data;

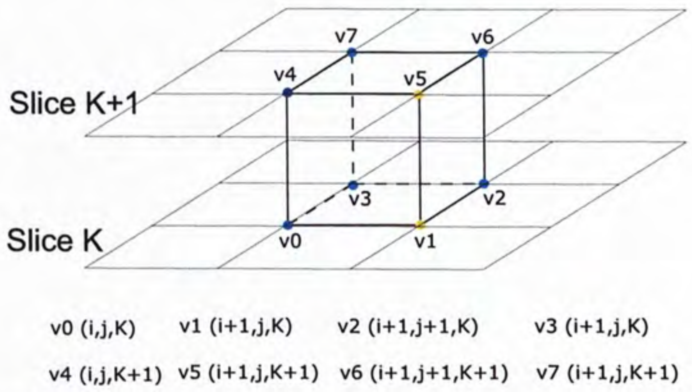


Figure 2.3: Each vertex is inside or outside the surface. Vertex is 0 when it is outside the surface; and 1 when it is inside the surface..

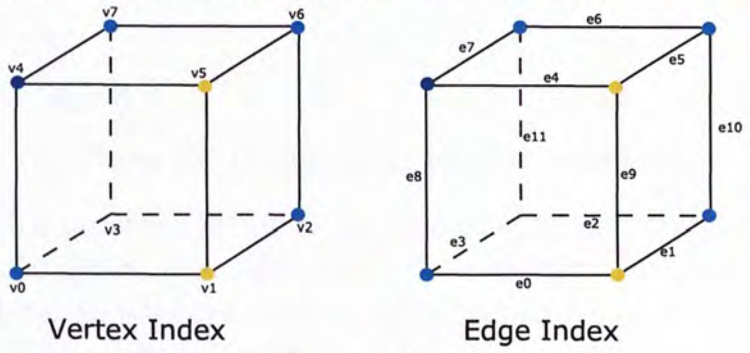


Figure 2.4: Vertex Index & Edge Index.

Comparison of 8 cube's vertices with threshold value; Index into triangle table creation (0-255); Normal vectors approximation in cube's vertices; Use of index to find all intersected edges of actual cube; Triangle vertices approximation at all intersected edges; Normal vectors approximation in triangle vertices.

Here we introduce the details of method.

1. Read four neighboring slices into memory as shown in Figure 2.5

2. The algorithm subdivides the whole volume into a series of cubes, and creates a cube from four neighbors on one slice and four neighbors on the next slice as shown in Figure 2.6.

3. The cube's vertices are classified surface value (1 for inside

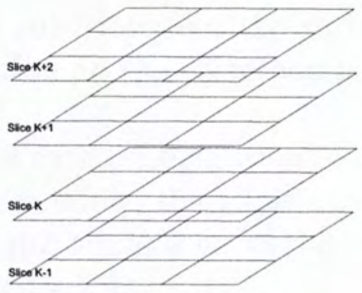


Figure 2.5: Four Slices.

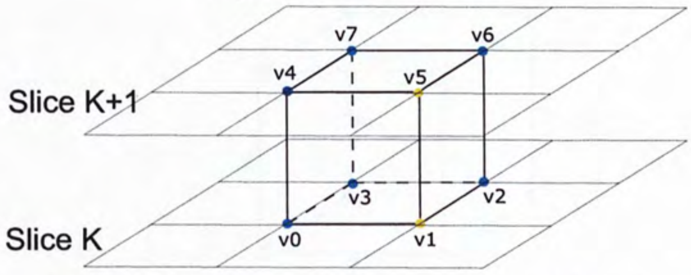


Figure 2.6: Creates a cube from four neighbors.

and 0 for outside vertices). By grouping these bits together in a specific order, the cube's index is calculated by the grouping these bits. we obtain a configuration type index (a value between 0 and 255) as shown in Figure 2.7.

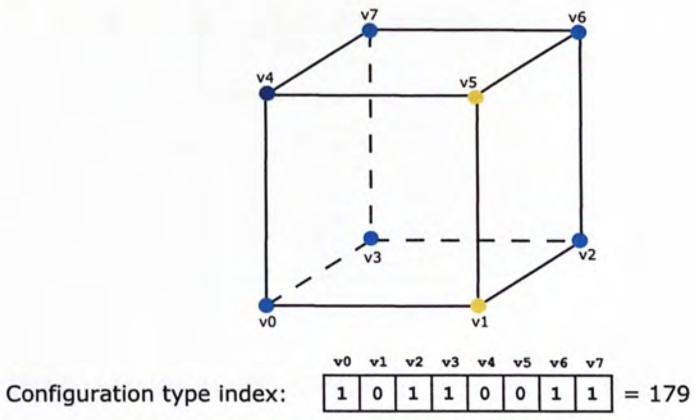
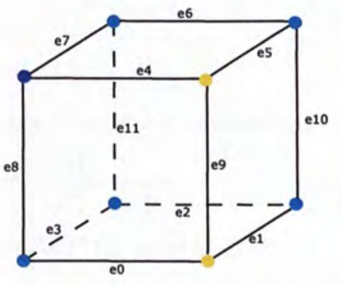


Figure 2.7: Calculate an index for the cube.

4. Use the index as a pointer into a 256-entry edge table, we look up the list of edges from a table. which is used to tell

which edges of the cube intersect the surface. Here, 1 means there is an intersection on the corresponding edge as shown in Figure 2.8.

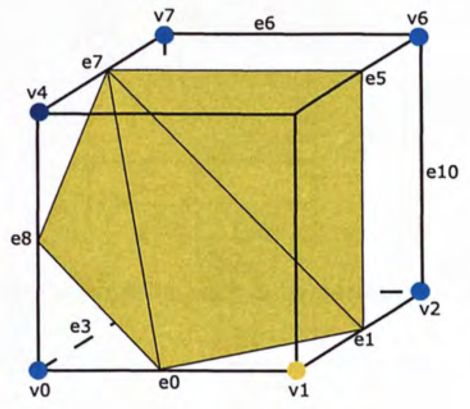
5.The triangles of each configuration is stored in a 256 entry triangle table. For example, the triple's index (0, 3, 6) means that the vertices of this triangle lie on the cube edges e0, e3 and e6 in this order. Figure 2.9.



Edge table (entry 179):

e0	e1	e2	e3	e4	e5	e6	e7	e8	e9	e10	e11
1	1	0	0	0	1	0	1	1	0	0	0

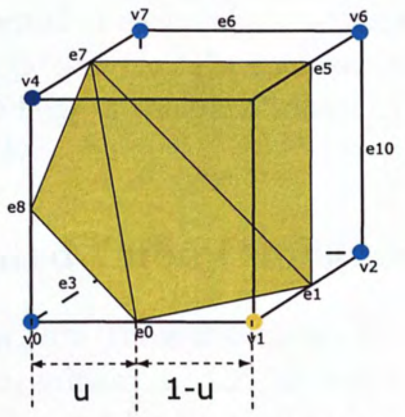
Figure 2.8: Look up the list of edges from a table.



Triangle table (entry 179): 0,1,7,1,5,7,0,7,8

Figure 2.9: Triangle table.

6. For each edge in the edge table, find vertex intersection position by linear interpolation as shown in Figure 2.10.

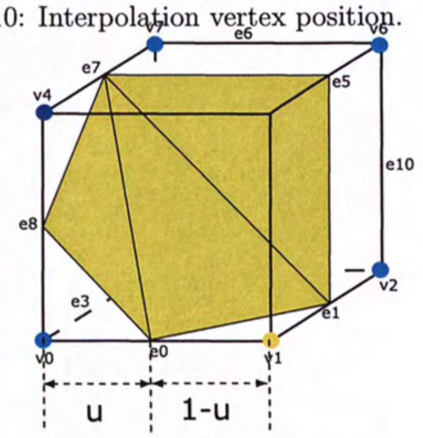


Interpolate surface intersection along each edge:

$$u = \frac{V0 - Vi}{V0 - V1}$$

$$Vi = V0*(1-u)+V1*u$$

Figure 2.10: Interpolation vertex position.



Calculate normal for each cube vertex:

$$\boldsymbol{G}x(i,j,k) = \frac{D(i+1,j,k) - D(i-1,j,k)}{\Delta x}$$

$$\boldsymbol{G}y(i,j,k) = \frac{D(i,j+1,k) - D(i,j-1,k)}{\Delta y}$$

$$\boldsymbol{G}z(i,j,k) = \frac{D(i,j,k+1) - D(i,j,k-1)}{\Delta z}$$

Interpolate the normals at the vertices of the triangles:

$$\vec{n}1 = \vec{g}0*(1-u)+\vec{g}1*u$$

Figure 2.11: Interpolation normal.

7. Calculate a normal at each cube's vertex and interpolate a normal to each triangle's vertex. For each vertex, find the vertex normal from the gradient of the data values by interpolation as shown in Figure 2.11.

2.3 Triangulated Cube Configuration Table

To simplify the algorithm, these 256 cases can be reduced to 15 patterns by rotation, mirroring, and inversion shows. Because there are eight vertices and two states(inside and outside of the isosurface), in each cube there are only $2^8 = 256$ possibilities of triangulated cube configuration that the isosurface can intersect the cube. The triangulation for the 15 patterns are listed, as follows:

Pattern 0

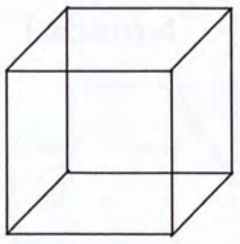
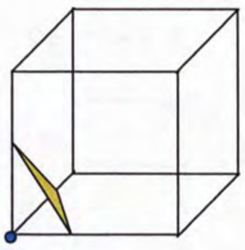


Figure 2.12:

Pattern 1



Rotation: C1, C2, C4, C8, C16, C32, C64, C128

Inverse: C127, C223, C239, C191, C247, C251, C253, C254

Figure 2.13:

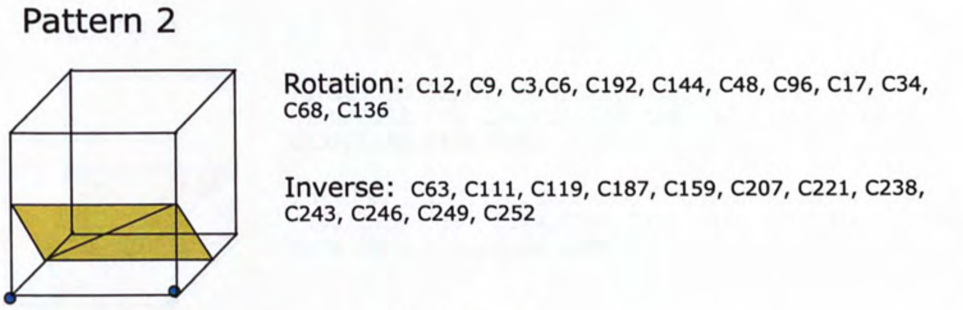

Pattern 2
Rotation: C12, C9, C3,C6, C192, C144, C48, C96, C17, C34, C68, C136
Inverse: C63, C111, C119, C187, C159, C207, C221, C238, C243, C246, C249, C252

Figure 2.14:

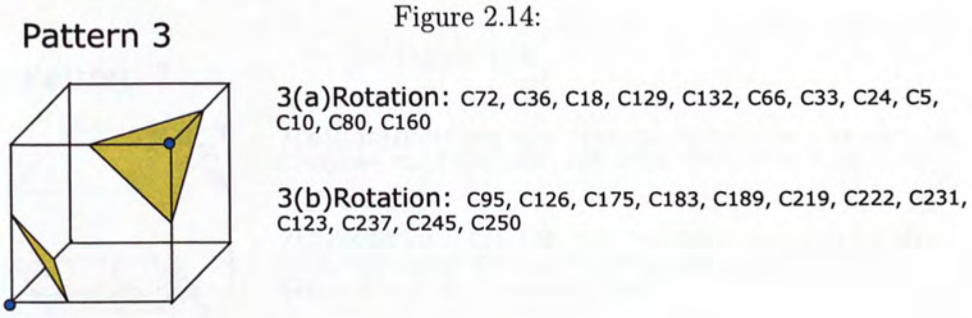

Pattern 3
3(a)Rotation: C72, C36, C18, C129, C132, C66, C33, C24, C5, C10, C80, C160
3(b)Rotation: C95, C126, C175, C183, C189, C219, C222, C231, C123, C237, C245, C250

Figure 2.15:

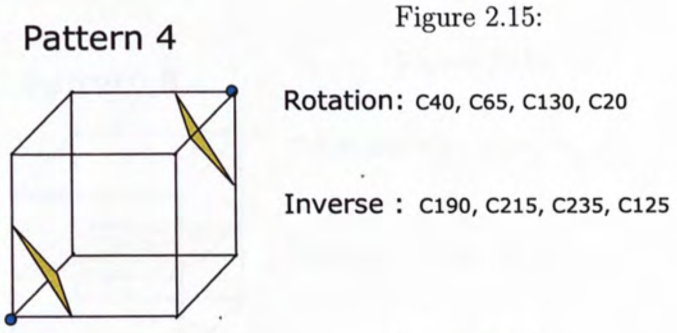

Pattern 4
Rotation: C40, C65, C130, C20
Inverse : C190, C215, C235, C125

Figure 2.16:

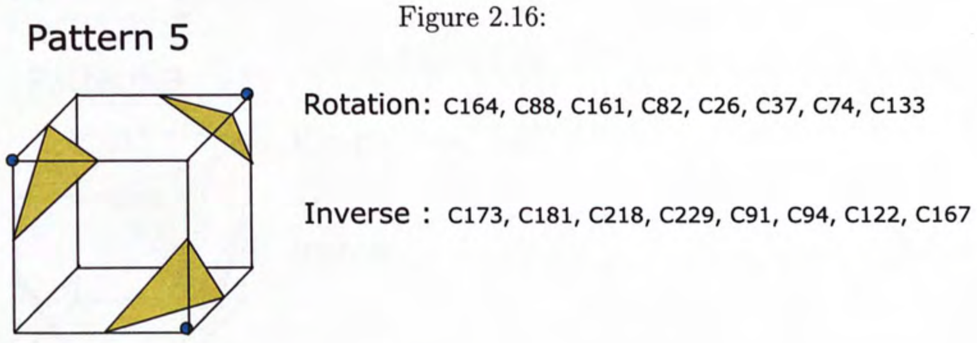
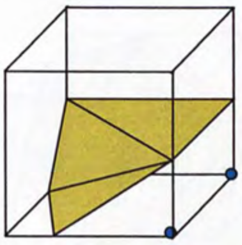

Pattern 5
Rotation: C164, C88, C161, C82, C26, C37, C74, C133
Inverse : C173, C181, C218, C229, C91, C94, C122, C167

Figure 2.17:

Pattern 6

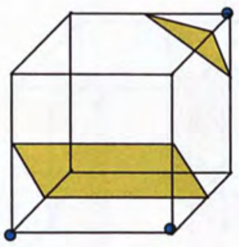


Rotation: C7, C11, C14, C13, C112, C176, C224, C208, C98, C196, C152, C49, C19, C25, C35, C38, C50, C70, C76, C100, C137, C140, C145, C200

Inverse : C31, C47, C55, C59, C79, C103, C115, C110, C118, C143, C155, C157, C179, C185, C205, C206, C217, C220, C230, C236, C241, C242, C244, C248

Figure 2.18:

Pattern 7

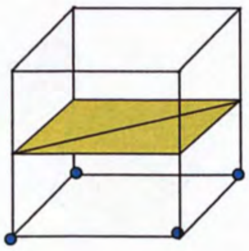


7(a)Rotation: C44, C73, C131, C22, C194, C148, C56, C97, C81, C162, C84, C168, C52, C67, C69, C104, C134, C138, C146, C193

7(b)Rotation: C61, C62, C87, C93, C107, C109, C117, C121, C124, C151, C158, C171, C174, C182, C186, C188, C199, C203, C211, C233, C234, C213, C214, C227

Figure 2.19:

Pattern 8

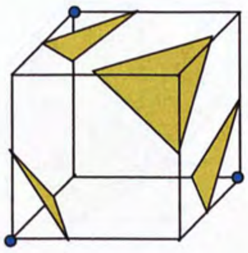


Rotation: C15, C102, C51

Inverse : C153, C204, C240

Figure 2.20:

Pattern 9

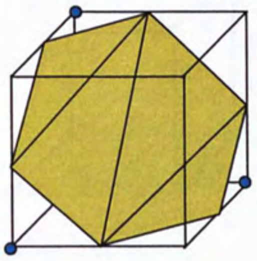


Rotation: C90, C165

Inverse :

Figure 2.21:

Pattern 10

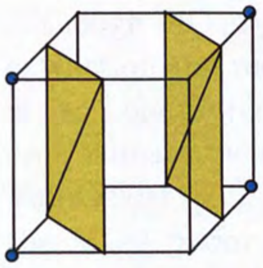


Rotation: C27, C39, C78, C141

Inverse : C114, C177, C216, C228

Figure 2.22:

Pattern 11

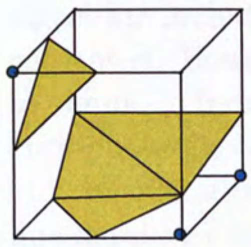


Rotation: C170, C60, C105

Inverse : C85, C195, C150

Figure 2.23:

Pattern 12

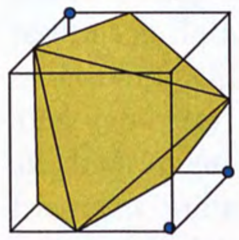


Rotation: C135, C75, C30, C45, C120, C180, C225, C210, C53, C58, C83, C86, C89, 92, C101, C106, C149, C154, C163, C166, C169, C172, C197, C202

Inverse :

Figure 2.24:

Pattern 13



Rotation: C23, C46, C29, C54, C57, C71, C77, C99, C108, C116, C113, C43

Inverse : CC232, C209, C226, C201, C198, C184, C178, C156, C147, C142, C139, C212

Figure 2.25:

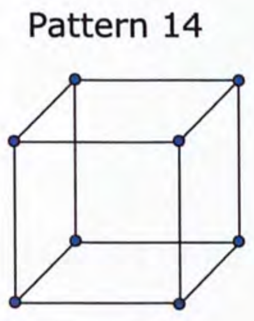


Figure 2.26:

2.4 Summary

Although MC algorithm has achieved great success in isosurface extraction and reconstruction, but has many problems: The one is the hole problem. This problem is caused by ambiguities in approximate the surface. This has been solved by Wilhelms and Van Gelder [72]. The second problem is about it's performance: the MC's performance has been improved by using octree to reduce the number of cubes traversed in Wilhelms and Van Gelder [73]. Other shortcomings of the original Marching Cubes algorithm include triangle quality, and large number of triangles generated. However, many basic MC smoothing techniques fail to eliminate terracing because their local neighborhood does not encompass the width of the terrace; and smoothing a mesh without consideration of the original data may smooth away crucial fine details as well as mesh generation artifacts.

However, the MC algorithm is still not adequate for interactive manipulation of 3D surfaces reconstructed from high-resolution data sets. Nowadays, in favor of the programmable function pipeline on the current GPUs, fully programmable parallel geometry and fragment units are available, via high level shading languages. In addition to computational functionality, fragment units also provide an effective memory interface to server-side data, i.e. texture buffer. In this work, we propose a

GPU-based Marching Cubes algorithm, which can speedup the time performance as well as avoid some disadvantages.

□ **End of chapter.**

Chapter 3

Graphics Processing Unit

3.1 Introduction

With the rapid progress in Graphics processing unit (GPU), various applications associated with computer graphics advance greatly. At the same time, the processing power, parallelism and programmability available on the current GPU provide an ideal platform for general-purpose computation such as algebraic computation, database operations and spectrum analysis. Starting from an introduction to the development history and the architecture of GPU, the technical fundamentals of GPU are described in this section. Some limitations of current GPU are also discussed. Many companies designed specialized and expensive graphics accelerator for transformation, rotation, illumination, rendering and texture mapping which are computationally intensive but necessary for almost all applications of computer graphics. In addition, these hardware are tailor made for ordinary fixed graphics pipeline. Therefore, they can only provide limited programming flexibility. Nowadays, the demand of graphics power is increasing. The primeval concept of graphics acceleration no longer meets the requirements. Consequently, programmable Graphics Processing Unit (GPU) is introduced. The key difference between such GPU and traditional graphic accelerator is the graphics hardware pipeline is broken from its hardwired elements into programmable pipelined processors [20]. The first impact of GPU is the realtime detailed and realistic cinematic graphics rendering. However, the user-level accessible parallel computation is the most important advantage we can perceive from programmable GPU. We can foresee that the power of parallel processing of GPU should be helpful for many complicated computational problem.

3.2 History of Graphics Processing Unit

Modern GPUs were designed from the monolithic graphics chips of the late 1970s and 1980s. At that time, a lot of chips are integrated together to handle complex computer graphics system. With the development of technology, these chips had limited BitBLT support in the form of sprites, and usually had no shape-drawing support. While, as we know, current GPUs can run several operations in a display list, and could use DMA to reduce the load on the host processor. Subsequently, hardware engineers integrated complicated multi chip design into a single graphics chip to enhance parallelism. For example, IBM introduced Video Graphics Array controller in 1987. At that time, VGA controller was only a simple hardware that dumps the output from CPU to the screen. In early 1990s, high-speed, general-purpose microprocessors became popular for implementing high-end GPUs. Several high-end graphics boards for PCs and computer workstations used TI's TMS340 series (a 32-bit CPU optimized for graphics applications, with a frame buffer controller on chip) to implement fast drawing functions. These were especially popular for CAD applications. In 1993, S3Graphics introduced the first single chip 2D accelerator, S3 86C911. After while, NVIDIA introduced "GPU", in late 1990s, as a term for VGA controller or 3D graphics accelerator to describe the graphics hardware. Contemporary GPUs include basic 2D acceleration and VGA frame buffer compatibility mode and most of the GPUs produced after 2000 support MPEG primitives, such as motion compensation and iDCT (inverse discrete cosine transform) [20].

3.2.1 First Generation GPU

In the first generation GPU were capable of rasterizing pre-transformed triangles and applying one or two textures. Two typical products of first generation of GPU were NVIDIA's TNT2 and ATI's Rage. The main problem of this type of GPUs is that it lacks of capability of vector and vertices transformation. As a result, the transformation of 3D object can solely be executed on CPU. Moreover, the number of texture access is limited in this generation GPUs.

3.2.2 Second Generation GPU

The second generation GPU appeared in late 1990s. Typical products are NVIDIA's GeForce 256 and ATI's Radeon 7500. The main feature of this type of GPUs is that it offers transformation and lighting. The fast hardware T&L transform offloads the CPU, which allowing much faster rendering process. Although a set of math operators for coloring pixels is supported, the limitation of this GPU is not fully programmable. Thus, users cannot design their own algorithm for special applications.

3.2.3 Third Generation GPU

In 2001, the third generation GPU included NVIDIA's GeForce3 and GeForce4 Ti, Microsoft's Xbox, and ATI's Radeon 8500. These types of GPUs can provide full vertex programmability rather than merely offering more configurability. But this generation of GPUs provides more pixel-level configurability but not programmability. The vertex-level programmability allows user to specify a program (sequence of commands) on a vertex. This is the main limitation of this type of GPUs. Many scientific research and image-based rendering are developed based on this type of GPUs.

3.2.4 Fourth Generation GPU

The fourth generation GPU included NVIDIA's GeForce FX family with CineFX architecture and ATI's Radeon 9700/9800. These GPUs support vertex-level and pixel-level programmability. The GeForce FX family even provides unlimited number of codes execute per rendering cycle. The fourth-generation GPUs consist of 280 million transistors. Based on some experiments and applications, this type of GPUs is able to draw about 540 million triangles per-second.

3.3 The Graphics Pipelining

3.3.1 Standard Graphics Pipeline

The graphics hardware processing is a fixed function pipeline to process the vertices, geometry, primitives and fragments. OpenGL is a graphics language that is designed as a streamlined, hardware-independent interface for different platforms. Nowadays it is referred as a standard graphics rendering pipeline. It consists of several different processing stages, including vertex transformation, assembly and rasterization, interpolation, texturing and coloring, and the final raster operations as shown in Figure 3.1.

Vertex transformation is the first stage of the pipeline, which is generating the position transform, texture coordinate generation and setting the lighting conditions. Then, the vertices will be transferred to the primitive assembly stage. The processing of this stage will assemble vertices into geometric primitives,

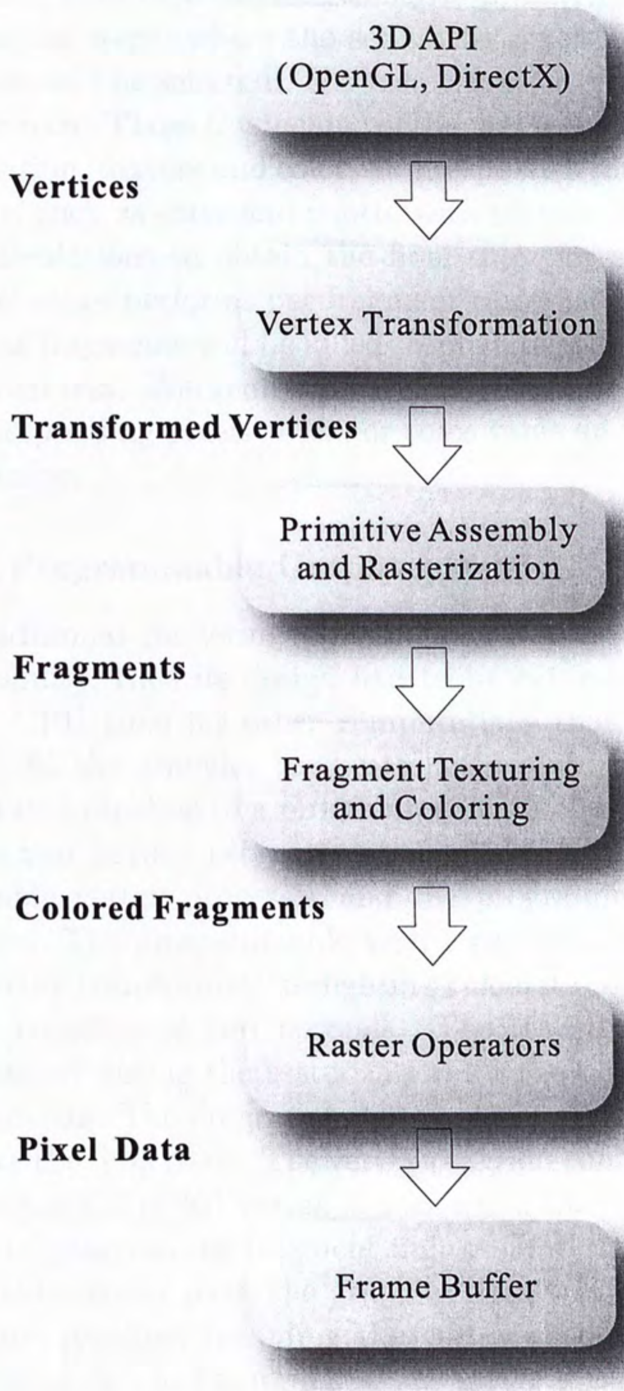


Figure 3.1: Graphics Pipeline

the points, lines or triangles geometric primitives flow into the rasterization steps, where the set of the pixels covered by the primitives will be selected. The result is a set of pixel locations on the screen. Those fragments will be processed at the stage of interpolation, texture and color. It interpolates the fragment parameters, such as color and depth, with texture looking up and math calculations to obtain the final color for each fragment. The final stage performs per-fragment rasterization operations, where the fragments will be killed through depths, scissor, alpha and stencil test. The remaining fragments will be blended with the corresponding pixels' alpha or color value and passed to the frame buffer.

3.3.2 Programmable Graphics Pipeline

The traditional rendering pipeline was not assigned for programmability; thus its design had to be extended, in order to free up CPU time for other computations than graphics processing. So the graphics hardware has evolved from a fixed or configurable pipeline to a programmable pipeline. This pipeline includes two distinct programmable processors, namely the programmable vertex processor and the programmable fragment processor. The programmable vertex processor is used to perform vertex transformation, lighting calculations, manipulating texture coordinates and normals. The transformed data will be processed during the rasterization for the positions and colors fragments. The programmable fragment processor is used to calculate the final color. The vertex program controlling the vertex processor is called vertex shader while the fragment shader is used to program the fragment unit of GPU. In a normal programmable render pass, the graphics data will be processed in the whole pipeline, including the vertex shader and fragment shader as shown in Figure 3.2.

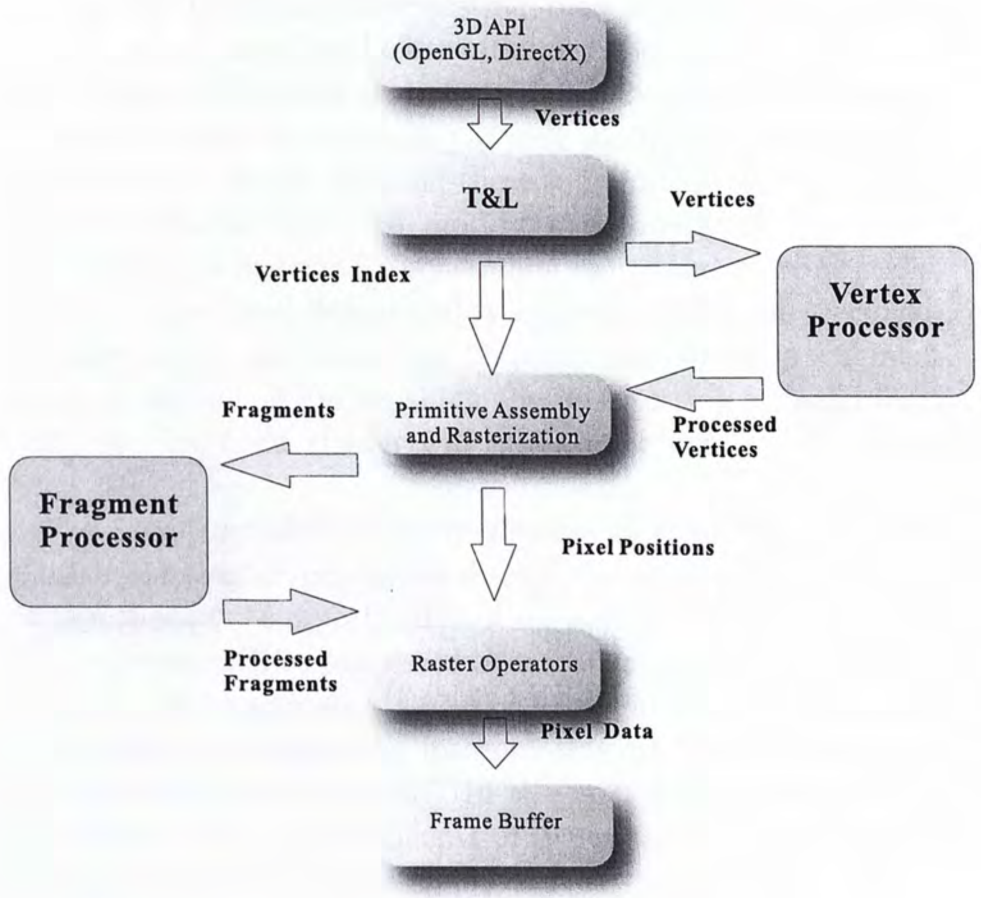


Figure 3.2: Programmable Graphics Pipeline

With the programmability, the shader can perform the texture fetching by looking up a specified texel value through a given texture coordinate. The texture coordinate can be obtained by interpolating from the vertex interpolation or by mathematically calculated in the shader. If the fragment is not killed, the results of the fragment shader are sent on for further processing. The remainder of the OpenGL pipeline remains as defined for fixed-function processing. Fragments are submitted to coverage application, pixel ownership testing, scissor testing, alpha testing, stencil testing, depth testing, blending, dithering, logical operations, and masking before ultimately being written into the frame buffer. The back end of the processing pipeline remains as fixed functionality because it is easy to implement in nonprogrammable hardware. Making these functions programmable is more complex because read/modify/write operations can introduce significant instruction scheduling issues and pipeline stalls. Most of these fixed functionality operations can be disabled, and alternative operations can be performed within a fragment shader if desired. After finishing the final testing for each fragment, the fragment shader will update the pixel in the frame buffer [19].

The programmable fragment processors require many math operations as vertex processors do [19]. Newer generation GPUs' texture operators support full floating-point values. Consequently, each fragment will be processed by running the fragment shader. The fragment shader should also be SIMD in nature. Final pixel value will be calculated by interpolating fragments color associate with the pixel location. In the next two subsections, we will introduce the basic concepts of the vertex processor and the fragment processor in details. [66]

3.3.3 Vertex Processors

The vertex processor is a programmable unit that offers the ability to directly control the operations for each vertex in the GPU. The vertex processor usually performs traditional graphics operations such as vertex transformation, normal transformation and normalization, texture coordinate generation, texture coordinate transformation, lighting and color material application. It replaces the transform and lighting operations of the fixed function pipelines for vertices in the traditional rendering pipeline, the vertex processor operates on one vertex at a time (but an implementation may have multiple vertex processors that operate in parallel). [66]

The design of the vertex processor is focused on the functionality needed to transform and light a single vertex. Several registers are associated with the vertex shaders. They are the input register, the output register, the constant register, temporary registers and address registers. The per-vertex input register is read-only for a vertex shader. The per-vertex data, like model-space vertex coordinates, vertex color and texture coordinates, are usually stored in the input register. While some attributes for the vertices which change per-frame or per-object, such as the transform matrices, or material properties, are contained in the constant register. The constant register is read-only also to the vertex shader. The temporal register assists the computation of the vertex shader. It is used to read and write the temporal result of the execution. A special temporal register, addressing register, is provided for the indirect addressing operations Output from the vertex shader is accomplished partly with special output variables. Vertex shaders must compute the homogeneous position of the coordinate in clip space and store the result in the special output variable "gl_Position". The result of the vertex shader execution is some predefined attributes, such as texture coordinates, color, clip-space vertex coordinates. These typical outputs are written to the output register. They later flow into the next stage of the graphics pipeline. [66]

3.3.4 Fragment Processors

The fragment processor is a programmable unit that capable of directly manipulating for each fragment in the graphics pipeline. The fragment processor usually performs traditional graphics operations such as operations on interpolated values, texture access, texture application, fog, and color sum. It replaces the texturing and coloring operations of the fixed function pipelines for fragment in the traditional rendering pipeline. One of the biggest advantages of the fragment processor is that it can access texture memory an arbitrary number of times and combine in arbitrary ways the values that it reads. A fragment shader is free to read multiple values from a single texture or multiple values from multiple textures. For each fragment, the fragment shader may compute color, depth, and arbitrary values (writing these values into the special output variables "gl_FragColor", "gl_FragDepth", and "gl_FragData") or completely discard the fragment. Several registers are associated with the fragment shader: the input register, output register, and temporal registers. The fragment shader can load texture as the input data. [66]

The geometric primitive has been rasterized into a set of fragments. It enters either the texture fetching stage or the fragment shading stage. Since fragment processors run in parallel, GPU has great speeding advantages over CPU. As native graphics application requires large amount of floating-point calculation and vector mathematics, GPU is specially designed for the vector type floating point operations which can run much faster than these of CPU. Figure 3.3 shows the basic procedure of fragment processor. It includes two components: texture fetching and fragment shading. [66]

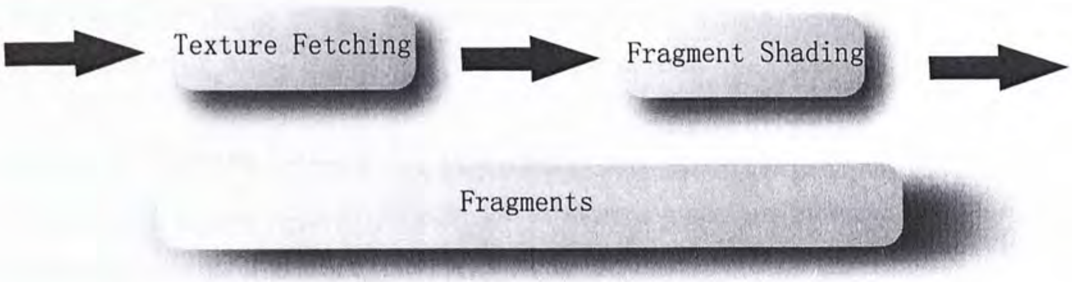


Figure 3.3: procedure of fragment processor

Texture Fetching Textures are 1D or multi-dimensional images that can be glued onto a 3D object. They are mapped onto geometric primitives in correspondence to the texture coordinates interpolated in the rasterization stage. This process yields an interpolated color value fetched from the texture. The order of interpolation is depending on the dimension of the texture target and the graphic hardware's capabilities. Current generation GPUs support the simultaneous fetching of multiple textures for each fragment without a hit in performance. Furthermore, these GPUs allow for enhanced controlling of the texture lookup itself. It is possible, for example, to use the color value returned by the first texture fetch as texture coordinates for consequent texture lookups. This is known as dependent texturing. Dependent texturing is important to implement different sorts of transfer functions for volume rendering. Other fragment attributes can be used as texture coordinates as well.

Fragment Shading The fragment shading stage applies further color operations on a given fragment to compute its final color. This stage is also capable of applying different math operations on a fragment's values. It may choose to change nearly every value of a fragment, e.g. the depth value, except for its screen location. Even allowing for the possibility that this stage may completely discard a fragment, it can thus prevent the fragment's corresponding screen pixel from being updated. The fragment shading stage emits one or more completely colored fragments for each input fragment it receives.

3.3.5 Frame Buffer Operations

The frame buffer operations stage performs a set of per-fragment operations right before the fragment is turned into an actual pixel. The incoming fragment is at first checked based on number of different tests. If any of these tests fail the pixel operations stage immediately discards the specific fragment without updating its corresponding pixel's value stored in the frame buffer. All tests can be enabled or disabled by the programmer, though it is not possible to change neither their order of sequence nor their functionality. If a fragment passes all the tests another set of operations is performed to update the values stored in the associated buffers. Thus the fragment has finally advanced to being a pixel. The sequence of frame buffer operations is illustrated in Figure ??.

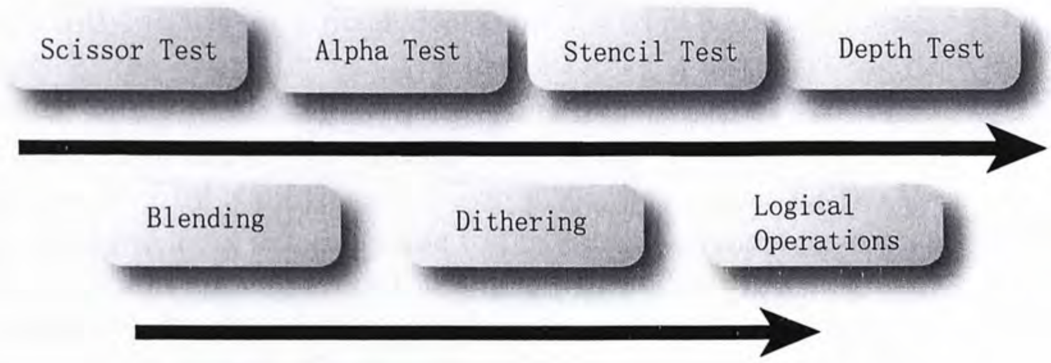


Figure 3.4: Fragment Shading

Scissor Test The scissor test is used to restrict drawing of pixels to a rectangular portion of the frame buffer. If a fragment lies inside this rectangle it is further processed by the subsequent operations.

Alpha Test The alpha test compares the incoming fragment's opacity, i.e. its alpha value, with a reference value. The fragment is accepted or rejected based on the outcome of this comparison.

Stencil Test The stencil test is typically used to mask out an irregularly shaped region of the frame buffer to prevent drawing from occurring within it. The pixel locations drawing is allowed or rejected on values stored in the stencil buffer that is part of the actual frame buffer. Therefore it resemblances the frame buffer in width and height. The stencil buffer is essential to the application of the stencil test, without it every fragment passes the stencil test automatically. The stencil test itself involves a comparison of the fragment's stencil value stored in the stencil buffer with a reference value. Optionally this comparison can also take the associated pixel's depth value into account. If fragment passes the stencil test it may choose to update the value stored in the stencil buffer as well.

Depth Test The distance between the camera origin and an object, i.e. the z-coordinate inside the view volume of an object, currently occupying a pixel location is stored in a specific buffer, namely the depth buffer. The depth buffer is also part of the frame buffer and extends to the same dimensions as the frame buffer. The depth test decides whether an incoming fragment is occluded by a previously drawn pixel, by comparing the incoming fragment's depth value to the associated pixel location's depth value already stored in the depth buffer. If a fragment passes the depth test it may choose to update the depth buffer value with its own. The depth buffer together with the depth test therefore provide a convenient mechanism for depth ordering either partially or fully occluded objects on a per-fragment level.

Blending After a fragment has passed all the pixel tests its color values are then combined with the color values already stored in the frame buffer at the corresponding location. This combination is referred to as blending. Different blending operations can be applied, such as replacing or modulating depending on the stored alpha values, thus allowing for semi-transparent objects.

Dithering By dithering, color resolution can be improved at the expense of spatial resolution, on systems with only a small number of color bit-planes. If the hardware already has a high color resolution the enabling of dithering will end up doing nothing at all.

Logical Operations The final operation on a fragment is a logical operation, such as OR, XOR, and NEGATE. This operation is applied before the fragment is written to the frame buffer, thus becoming a pixel, to the incoming fragment's values and/or the values currently stored in frame buffer. [46]

3.4 GPU CPU Analogy

The CPU in a modern computer system communicates with the GPU through a graphics connector such as a PCI Express or AGP slot on the motherboard. Because the graphics connector is responsible for transferring all commands, textures, and vertex data from the CPU to the GPU, the bus technology has evolved alongside GPUs over the past few years. The original AGP slot ran at 66 MHz and was 32 bits wide, providing a transfer rate of 264 MB/sec. AGP 2, 4, and 8 followed, each doubling the available bandwidth, until finally the PCI Express standard was introduced in 2004, with a maximum theoretical bandwidth of 4 GB/sec simultaneously available to and from the GPU. GPU is a stream processor while CPU is a serial von Neumann architecture. Therefore, the underlying methods of processing of GPU and CPU are totally different. There are some constraints should be applied to GPUs, thus not every program can be mapped onto the GPUs. In this section, we will discuss the two fundamental conceptual differences between GPU and CPU: memory architecture and processing model.

3.4.1 Memory Architecture

GPUs use standard DRAM modules rather than custom RAM technologies to take advantage of market economies and thereby reduce cost. Having smaller, independent memory partitions allows the memory subsystem to operate efficiently regardless of whether large or small blocks of data are transferred. All rendered surfaces are stored in the DRAMs, while textures and input data can be stored in the DRAMs or in system memory. The four independent memory partitions give the GPU a wide (256 bits), flexible memory subsystem, allowing for streaming of relatively small (32-byte) memory accesses at near the 35 GB/sec physical limit. The memory on GPU is textures. A texture can be considered as a 2D array of memory texels with limited size constraint, and each texel can have either 1, 3 or 4 channels. Like main memory, each texel has a texture coordinate, and the value in case of 3 or 4 channels stored can be accessed directly. It is not as flexible as arrays; however, the three or four color channels design makes it is a perfect data structure for storing vector components of scientific computation. Operations on memory are specially designed for multi-channel architecture. The cost for operations on multi-component is approximately same as the operations on single component. Because the memory accessing speed of GPU is generally faster than main memory, fetching data stored in GPU memory is considerably much faster.

3.4.2 Processing Model

The processing model of GPUs is totally different from CPU. The former is stream processor while the latter is serial processor. The most essential difference between stream processor and serial processor is that every object in the stream processor is processed by the same function. The texture and fragment-processing unit operates on squares of four pixels (called quads) at a time, allowing for direct computation of derivatives for calculating texture level of detail. Furthermore, the fragment processor works on groups of hundreds of pixels at a time in single-instruction, multiple-data (SIMD) fashion (with each fragment processor engine working on one fragment concurrently), hiding the latency of texture fetch from the computational performance of the fragment processor. Each fragment executes the identity fragment program simultaneously and independently. Another important difference is that the program in stream processor is limited to undetermined looping or branching.

3.4.3 Limitation of GPU

Programmable GPUs have a higher computational power than CPUs, because they are explicitly designed for the simultaneous processing of multiple data-parallel primitives. However, compared to CPUs they offer only a limited instruction set consisting primarily of mathematics operations which are often graphics specific and in general accept as input a limited number of 32-bit floating point 4-vectors. The vertex stage can output a limited number of these floating point vectors, which are interpolated by the rasteriser and passed as input vectors to the fragment stage. Currently the fragment processor can output only 4 floating point 4-vectors, usually representing colors. Each programmable stage has access to global constants and local temporary registers. Since the write position of a processed fragment is determined in advance by the vertex-parameters and cannot be changed within a fragment program, fragment processors are incapable of performing memory scatter. It is possible to perform memory scatter operations via vertex programs through the recently emerged vertex-texture-fetch capability of current GPUs and the vertex processors' ability to change the target memory address of the colored fragments. This, however, can lead to memory and rasterisation coherence issues and lower performance.

3.4.4 Input and Output

With GPU, we mainly use fragment program to perform calculations. Normally, textures are used as input. There is a size limitation on the textures, GeForceFX series support maximal size of 4096 X 4096 texture data or 512 X 512 X 512 volume data. The total number of textures being accessed simultaneously is limited. Moreover, the input textures cannot be used as output. The output of a fragment is limited to a single output vector. As a result, shader programs can only have a single output stream. For the problems with large input and output, several rendering passes are usually needed.

3.4.5 Data Readback

Readback is one of the biggest limitations for computation on GPU. Today the data transfer from the GPU to the CPU becomes a bottleneck. The transferring rate from GPU to CPU is very slow compared with the GPU memory accessing speed. For sequential processing, data readback from GPU to CPU is unavoidable. To avoid this penalty, computation must be performed on GPU as much as possible to avoid readback.

3.4.6 Framebuffer

The framebuffer is a video output device that drives a video display from a memory buffer containing a complete frame of data. The information in the buffer typically consists of color values for every pixel (point that can be displayed) on the screen. Color values are commonly stored in 1-bit monochrome, 4-bit palletized, 8-bit palletized, 16-bit highcolor and 24-bit truecolor formats. An additional alpha channel is sometimes used to retain information about pixel transparency. The total amount of the memory required to drive the framebuffer is dependent on the resolution of the output signal, as well as the color depth and palette size.

This framebuffer extension defines a new OpenGL object type, called a "renderbuffer", which encapsulates a single 2D pixel image. The image of renderbuffer can be used as a framebuffer-attachable image for generalized offscreen rendering and it also provides a means to support rendering to GL logical buffer types which have no corresponding texture format (stencil, accum, etc). A renderbuffer is similar to a texture in that both renderbuffers and textures can be independently allocated and shared among multiple contexts. The framework defined by this extension is general enough that support for attaching images from GL objects other than textures and renderbuffers could be added by layered extensions. [60]

3.5 Summary

The power of programmable GPUs enables efficient computation of a wide variety of applications. It used to enhance the visual appearance of interactive 3D rendering and accelerate the rendering process. But exploiting the efficient parallel performance of GPU, it also has the ability to perform varieties of general purpose computation.The general purpose applications include data set operations [62] [14] [4] [53] [49] [13] ,collision detection [30] [51], computational geometry [7] [51] [29] [52] [68] [1], scientific computing such like fluid simulation [75], cluster [18], matrix multiplication [42] [28] [31], physical simulation [45] [50] [35]and FFT [47] .

In this thesis, we will demonstrate that GPU can do more than just rendering.Our approach will exploits OpenGL framebuffer object [62] to store the rendered texture and/or transfer it from GPU framebuffer to CPU memory. The parallelism makes it a fast platform to handle many computer graphic problems as well as other general computational problems. However, there are limitations on GPU for its stream programming model. This motivates us to rethink how we solve certain problems. As GPUs continue to grow at a rapid pace, it is likely that GPU is becoming a mainstream for general-purpose computation.

□ **End of chapter.**

Chapter 4

Volume Rendering

4.1 Introduction

Volume rendering has become a large part of scientific visualization during the last twenty years. It is a technique used to display a 2D projection of a 3D volumetric data set. 3D scalar fields are generated within a wide range of scientific areas and visualization of data is important to quickly and accurately gain insight to large amounts of information. This type of data cannot be rendered with conventional rendering techniques, which is the reason that volume rendering has created its own field within scientific visualization. Rendering a volume is a computationally intensive task due to the large amount of data that need to be processed, and it is only recently, with the advent of commodity 3D hardware accelerator cards, that interactive rendering of volumes has become possible. So this field has a number of applications, especially within medical imaging, where the output of CT and MRI scanners is a volume data set, as well as geology where seismic surveys are visualized as an aid when searching for oil or gas.

The three basic principles of volume rendering are forming of an RGBA volume from the volume data, reconstruction of a function from this discrete volume data set, and projecting it onto the 2D viewing plane (the output rendered image) from the point of view. so volume rendering is the process of transforming a set of 3D discrete sample color points to a 2D image which can be displayed on a screen. as a color volume is a 3D four components RGBA volume data set, where the first three components R, G, and B color components and the last A components opacity. 0 is opacity value that means totally translucent, 1 is opacity value means totally opaque. the background color is placed, Behind the color volume is an opaque. The classification of the data as opacity values is mapped by the alpha table. The appearance of isosurfaces can be improved by using shading techniques to form the Raycasting, The size of a volume data is increased by the lengths of the volume elements. Even relatively small volumes usually contain a significant number of samples, more commonly called voxel for volume elements. For the implementation of a volume renderer, several methods are utilized, mostly depending on the available hardware. If the hardware is recent enough to support 3D texture mapping, which, while slightly more computationally intensive, requires less texture memory, interactive performance can be achieved. Further more, it can generate better visual quality.

In this chapter, after a brief introduction of the development of volume rendering techniques, we focus on various hardware-accelerated volume rendering techniques. The principles of some key techniques are explained in detail. Finally, we will give a short summary.

4.2 History of Volume Rendering

In the 1970's, volume rendering techniques have been developed to enable more direct visualization of the volumetric data. With appropriate preprocessing, volume rendering can be used to visualize surfaces, interior structure, and objects that do not have well defined surfaces. In 1990, Kaufman et al. developed many effective volume editing tools, volume rendering algorithms and data compression schemes [33]. Although the process needs large memory, faster algorithms and special-purpose hardware are enabling realtime volume rendering of data of significant size and resolution.

In 1993, Kaufman et al. have introduced the field of volume graphics, where a voxel-based data format is used to represent graphical objects, which are customarily represented by surface-based models [34]. They have demonstrated that many of the graphical effects, such as shading and reflectance that are available in surface-based graphics representation are also possible using volume graphics. Assuming that memory needs and processing requirements can be met effectively, Kaufman et al. asserted that volume graphics has the potential to supersede surface-based graphics just as 2D raster graphics superseded vector graphics [34]. Whether or not this potential is realized will depend on many factors. However, some objects will be more accurately modeled using a voxel based volume graphics format than conventional graphics formats.

Cullip and Neumann [12] discussed the necessary sampling schemes as well as object-aligned and view-aligned sampling planes in 1993. Based on this idea, as well as the extension to more advanced medical imaging, a novel technique was described by Cabral et al. [8]. They demonstrated that both interactive volume reconstruction and interactive volume rendering was possible with hardware providing 3D texture acceleration.

In 2001, K. Engel et al.implemented the high-quality pre-integrated volume rendering using hardware-accelerated pixel shading. Then the authors also described the interactive high-quality volume rendering based on flexible consumer graphics hardware [16]. In 2003, Roettger et al. described a GPU-based pre-integrated texture-slicing including advanced lighting. In the same year Krger and Westermann proposed a method to accelerate volume rendering based on early ray termination and space-skipping in a GPU-based raycasting approach [41]. The space-skipping addresses the rasterization bottleneck, using a single octree level only.

Today, by exploiting the capabilities of current hardware, volume rendering approaches using textures have become more and more popular. Based on these hardware accelerated algorithms, manipulations of volumes can be performed in real time.

4.3 Hardware Accelerated Volume Rendering

Volume rendering techniques based on graphics hardware utilize texture memory to store a 3D data set. Current graphics cards have become programmable with high-level shading languages which allow them to execute small programs for each pixel in the final image. Their architecture is highly parallel with 16 or even 24 pixel pipelines working concurrently. The volume to be displayed is restricted by the available amount of texture memory and the transfer rate between main and graphic memory. Volume rendering techniques based on texture memory include two types of texture-based techniques, 2D texturing and 3D texturing.

The first one is 2D texture-based slicing technique, which is along the major axes of the data and takes advantage of hardware bilinear interpolation within the slice. These methods require three copies of the volume to reside in texture memory, one per axis, and they often suffer from artifacts caused by undersampling along the slice axis. Trilinear interpolation can be attained using 2D textures with specialized hardware extensions available on some commodity graphics cards. This technique allows intermediate slices along the slice axis to be computed in hardware. These hardware extensions also permit diffuse shaded volumes to be rendered at interactive performance [9].

The other one is 3D texture-based techniques, which typically samples view-aligned slices through the volume, leveraging hardware trilinear interpolation. Other proxy geometry, such as spherical shells, may be used with 3D texture methods to eliminate artifacts caused by perspective projection. The pixel texture OpenGL extension has been used with 3D texture techniques to encode both data value and a diffuse illumination parameter which allows shading and classification to occur in the same lookup. Engel et al [16]. showed how to significantly reduce the number of slices needed to adequately sample a scalar volume, while maintaining a high quality rendering.

4.3.1 Hardware Acceleration Volume Rendering Methods

The OpenGL application programming interface provides access to the advanced per pixel operations that can be applied at the rasterization stage of the graphics pipeline, and in the frame buffer hardware of modern graphics workstations. During this process the volume data set is then sampled, classified, rendered to proxy geometry, and composited. Classification typically occurs in hardware as a 1D table lookup.

In particular, they provide sufficient power to render high-resolution volume data sets with interactive frame rates using 2D or 3D texture mapping. When using texture hardware to render volumes, volume is sliced in one of two ways. The first mode of slicing is object-aligned slicing. With object-aligned slicing, the slices are fixed to the volume, much like in shear-warp factorization. Because the slices are fixed with respect to the volume, the data for each slice may be stored in a 2D texture. Of course, slices will not be visible if they are parallel to the viewing direction. For object-aligned-slices, there must be at least three copies of the slices where each set is perpendicular to a principle axis of the volume. The second mode of slicing is view-aligned slicing. With view-aligned slicing, the slices are always perpendicular to the view plane and the renderer trilinearly interpolates the volume data to map onto the slice.

4.3.2 Proxy Geometry

The first step of GPU-accelerated volume rendering is to place geometry inside the three-dimensional scalar field that constitutes the volume. A set of texture coordinates are interpolated along the surface of the geometric primitive as well as other attributes when this geometry is rendered. Each generated fragment is assigned its corresponding set of texture coordinates in the rasterization stage. This set of texture coordinates can later be used to sample one or several texture maps at the associated location. Subsequently, to sample the volume at arbitrary locations, the scalar field constituting the volume must be stored in one or several textures while the texture coordinates must be assigned to correspond to locations inside this scalar field. The geometry does not have any relations to the data contained in the volume. That is why it is called "proxy geometry".

The proxy geometry characterization step in the graphical pipeline can be specified by two methods. The first one is enclosing rectangles of intersections while the other is enclosing polygons of intersections. The former is a straightforward method of texture mapping cut-planes. The latter requires finding the polygon of intersection between a given cut-plane and the cube of data. This approach is relatively faster for processing fragments because one visits only those fragments that are inside the cube of data. It is proposed by Kniss et al [39]. including the following key steps: (1) transform the volume bounding box vertices into view coordinates; (2) find the minimum and maximum z coordinates of the transformed vertices; (3)for each plane, in back-to-front order test for intersections with the edges of the bounding box and add each intersection point (up to six) to a fixed-size temporary vertex list; (4)compute the centre of the proxy polygon by averaging the intersection points and sort the polygon vertices clockwise;(5) tessellate the proxy polygon into triangles and add the resulting vertices to the output vertex array.

4.3.3 Object-Aligned Slicing

2D texture mapping is well suitable for implementing object-aligned slicing. It involves storing a set of three rectilinear volume data sets, and using them as three perpendicular stacks of object aligned texture slices (Figure 4.1). Slices are taken through the volume orthogonal to each of the principal axes. The resulting information for each slice is represented as a 2D texture that is then pasted onto a square polygon of the same size. The rendering is performed by projecting the textured quads and blending them back-to-front into the frame buffer. During the process of texture mapping the volume data is bilinear interpolated onto a slice polygon.Figure 4.1 shows an example using 2D texture mapping to render a volume human head dataset.

As mentioned above, a single stack of 2D slices is not enough for visualization of the volume. It would be possible to see through the individual slices when the point of view is rotated around the textured proxy geometry during rendering. This problem cannot be accounted for with just a single stack of slices. This is why three stacks of slices must be stored. In general, the stack with slices most parallel to the screen plane is chosen for rendering the volume as illustrated in Figure 4.2.

The main limitation of using object-aligned slices for volume

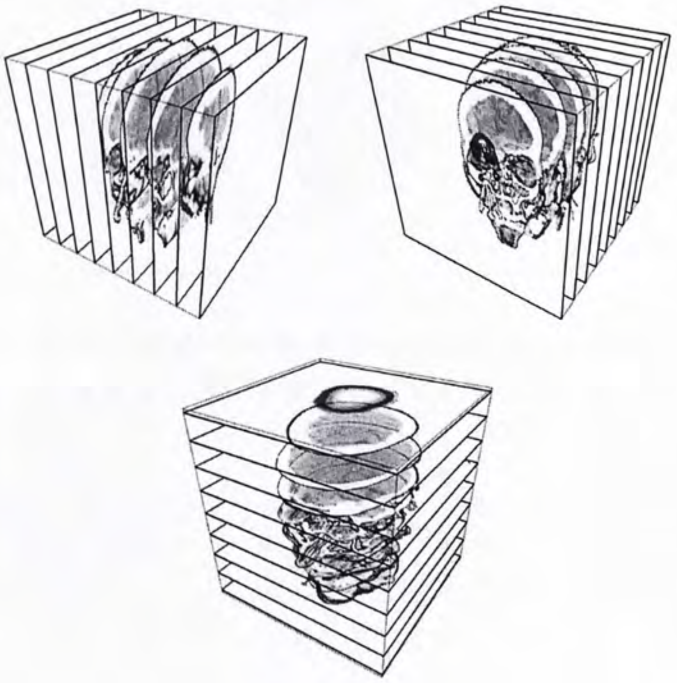


Figure 4.1: Object-Aligned Slicing.

visualization is the space performance since the requirement for three slice stacks that allocate three times the memory than the actual volume. Another drawback is that the switching of the stack currently used for rendering yields visible artifacts and a sudden drop in rendering performance. Also, when one stack is switched to another, artifacts can become visible. This is because that the actual locations of re-sampling points change abruptly with the change in stacks as illustrated in Figure 4.3.

Another limitation of object-aligned slice is that using this method leads to inconsistent sampling rates for different viewing directions. This is because alpha blending is used to accumulate the re-sampled values and performing a numerical integration of the volume rendering integral. The same effect is achieved using this method as composting samples along a ray in ray-casting. The sampling distance is dependent on distance between adja-

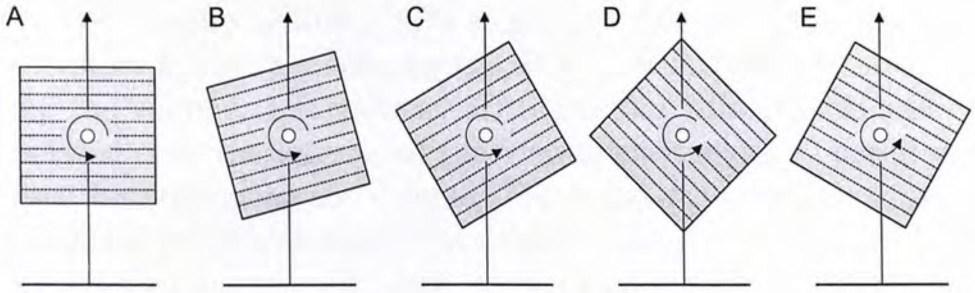


Figure 4.2: Choose the stack with slices most parallel to the screen plane.

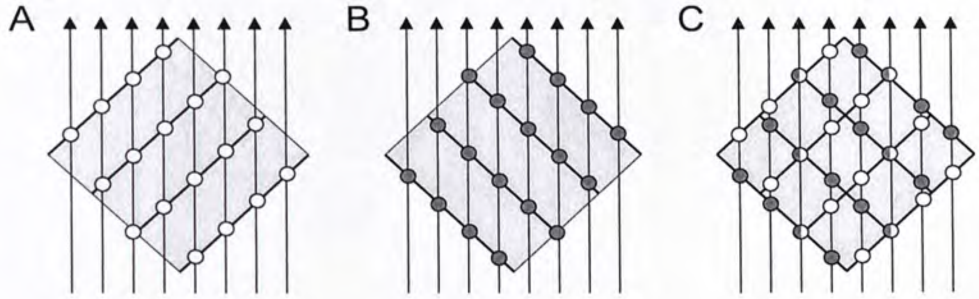


Figure 4.3: Visible artifacts caused by switching of the stacks .

cent object-aligned slices.

4.3.4 View-Aligned Slicing

Recent years, 3D texture mapping hardware has become a powerful visualization option for interactive volume rendering. This method is usually executed in the following steps: (1) convert volume data to a 3D texture; (2) a number of planes perpendicular to the viewer's direction of sight are clipped against the volume bounding box; (3) the texture coordinates in parametric object space are assigned to each vertex of the clipped polygons. The main difference between view-aligned slicing and object-aligned slicing is that during rasterization, fragments in the slice are trilinearly interpolated from 3D texture and projected onto the screen plane using adequate blending operations. Figure 4.4 shows the basic principle of this approach.

The volume is stored in a single 3D texture. View-aligned slices are used to generate re-sampling locations for reconstructing the volume. In this case, 3D texture coordinates are interpolated over the interior of the view-aligned slices. Then it is used for addressing the volume. This approach takes advantage of spatial coherence inside the volume.

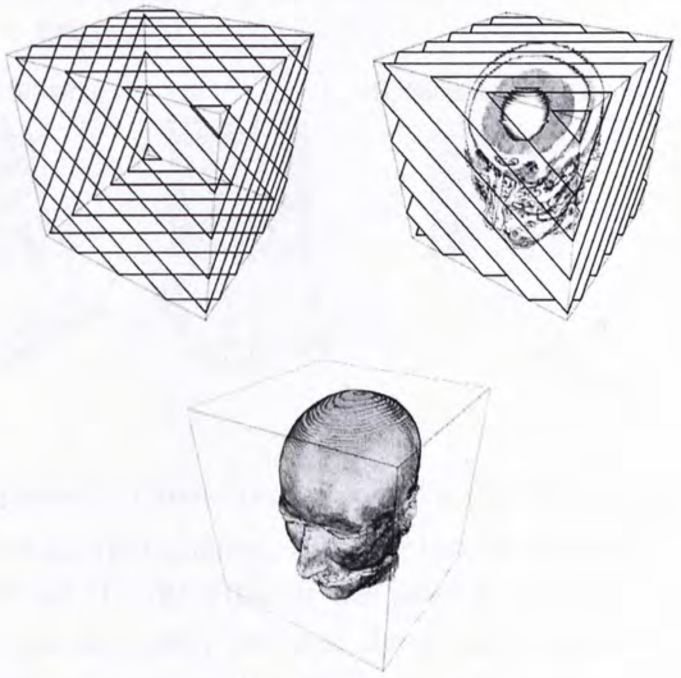


Figure 4.4: View-Aligned Slicing.

One of the major advantages of using 3D texture is that slices can be oriented arbitrarily with respect to the volume. This allows arbitrary orientation of slices to be used for re-sampling the volume. View-aligned slices are rendered as proxy geometry so that ray-casting can be mimicked as close as possible to the image plane. Especially, this approach offers an equidistant sampling rate for all viewing directions for orthogonal projection, mimicking ray casting perfectly for each "ray", i.e. for each final pixel (Figure 4.5 (1)). However, in case of perspective projection, the distance between successive re-sampling locations is not equal to adjacent "rays", i.e. for adjacent pixels (Figure 4.5(2)). Although this approach offers a good approximation of the final result, it is possible to render spherical shells as proxy geometry. This offers an equidistant sampling distance for perspective projection at the expense of more vertices that need to be processed.

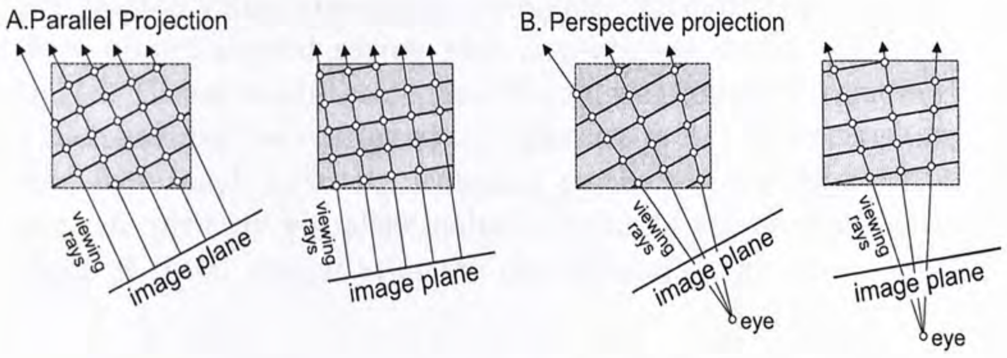


Figure 4.5: Parallel projection and perspective projection.

In view-aligned slicing, the number of slices can be chosen arbitrarily on-the-fly without the need for setting up inter-slice interpolation manually because the graphics hardware performs general trilinear interpolation for each fragment during the re-sampling process. This is a major advantage of using 3D textured view-aligned slices, as it results in an image of higher quality. In addition, it is also possible to render slices with arbitrary orientation with respect to the volumetric data which allows maintaining a constant distance between sampling points for all pixels and viewing directions. Furthermore, a single 3D texture allocates only a third of the memory that the three object-aligned 2D textures stacks do. The major disadvantage is that trilinear interpolation is significantly slower than bilinear interpolation, due to the requirement for using eight as opposed to four texels for each computed sample. Another drawback of this approach is it requires texture fetch patterns that decrease the efficiency of texture caches on graphics memory.

4.4 Summary

In this chapter, we give a detailed introduction of existing volume rendering techniques. Developed from the 1970s, volume rendering technique had becoming a large part of computer graphics, especially in scientific visualization. Several hardware-accelerated volume rendering techniques, including proxy geometry, object-aligned slicing and view-aligned slicing are introduced. Based on object-aligned slicing, we propose a framework which ensures the extracted triangles are drawn in a correct order, from back to front, according to the viewing direction as well as correctly visualize multiple layers of translucent isosurfaces, without computationally expensive sorting.

□ **End of chapter.**

Chapter 5

GPU-Friendly Marching Cubes

5.1 Introduction

Isosurfaces have been widely adopted to reveal the complex structures in medical and scientific volume data. In SIGGRAPH 1987, Marching Cubes (MC) was presented by Lorensen and Cline. From then on Marching Cubes is the most commonly-used algorithm for finding polygonal representations of isosurfaces in 3D volumetric data. Isosurfaces extraction is a common analysis and visualization technique for three-dimensional scalar data due to its fine visual quality. Visualizing multiple layers of translucent isosurfaces (normally represented as triangles) not only just generates high-quality rendering results, but also allows viewers to better understand the relationship among internal structures. However, visibility-correct visualization of multiple translucent isosurfaces imposes a lot of difficulties. Standard depth-buffering alone cannot resolve the visibility of overlapped translucent triangles. Extracted triangles can be sorted by depth sorting or binary space partitioning (BSP) based visibility sorting in order to generate a correct drawing order. In other words, computational expensive sorting has to be performed whenever the viewpoint changes.

The Marching Cubes algorithm have two basic parts : the triangulation module and rendering module. In the current MC algorithm, the triangulation module, which needs more arithmetic and logical operations, is run on the CPU. The rendering module sends the vertices' positions and normal to GPU for display. Nowadays, in favor of the programmable function pipeline on the current GPUs, fully programmable parallel geometry and fragment units are available, via high level shading languages. In addition to computational functionality, fragment units also provide an effective memory interface to server-side data, i.e. texture buffer.

So we present a GPU-friendly MC implementation. Besides the cell indexing, we propose to calculate vertex and normal interpolations by precomputing the expensive equations and looking up these values during runtime. Upon a commodity GPU, our implementation can rapidly extract isosurfaces from a high resolution volume and render the result. With the proposed GPU based Marching Cubes algorithm, we can naturally generate layer structured triangles, which facilitate the visibility-correct visualization of multiple-layer translucent isosurfaces without performing computational expensive sorting. The algorithm extracts and draws triangles, in a layer by layer fashion, from back to front. With the proposed algorithm, the visibility of multi- layer translucent isosurfaces is resolved naturally.

5.2 Previous Work

Surface reconstruction is widely applied for volume rendering [32]. In [74], Xie *et al.* presented an algorithm that recovers surfaces from noisy and defective data, by fitting surface in each octree cell. Nilsson *et al.* [59] reconstructed 3D closed surfaces from parallel contours. Isosurface is important for visualizing complex structures, especially for 3D medical data, such as ultrasound, CT, and MRI scans. Marching Cubes is one of the most famous techniques for extracting isosurfaces, first developed by Lorenson and Cline [44]. Later its variants [10, 43, 54, 55, 56, 57, 58] were proposed to solve some problems, such as ambiguity, accuracy, efficiency, and so on. Instead of cubes, some extensions employ various substitopes [3], such as tetrahedra [15, 24] and diamonds [2].

Bertram [5] proposed a technique for volume refinement by fairing isosurfaces. In [23], Gregorski *et al.* presented an algorithm for interactively extracting and rendering isosurfaces of large volume data set in a view-dependent fashion. Gerstner and Pajarola [21] described a technique of multiresolution isosurface visualization. In [25], Guo presented a method of isosurface extraction by interval set. ISOSLIDER [11] interactively displays the updated isosurfaces as the isovalue is slightly changed. The fast update exploits the coherence of isosurfaces with similar isovalues.

To take advantage of the new programmable graphics chips, many techniques are developed to speed up algorithms previously run on the central processing unit (CPU). The intrinsic parallelism computation and memory communication on a GPU have been exploited to accelerate the general-purpose computation [63], for example FFT [48]. Moreover, simulations of computer graphics techniques can be accelerated by graphics hardware, including particle systems [36, 40], collision detection [22], fluid dynamics [27], global illumination [65], ray tracing [70], and so on.

Especially, there are some hardware-accelerated techniques for volume rendering. The technique proposed in [71] utilizes programmable graphics hardware to accelerate volume visualization, by per-pixel operations available in the rasterization stage and in the frame buffer hardware. Binotto *et al.* [6] proposed a volume rendering technique using a fragment-shader compression. In [26], Hadwiger *et al.* presented a GPU-based ray casting and advanced shading of isosurfaces. However, their work loses one important property of isosurfaces: the geometry. They just visualize the isosurfaces without actually constructing the geometric structure. Based on the Marching Tetrahedra (MT) algorithm, [64] and [67] utilize vertex shader to look up tetrahedra and finally render the extracted surfaces. These two methods only reconstruct triangles temporarily and cannot completely store isosurfaces (triangular mesh). Recently, in favor of the more powerful fragment shader, several methods, which can store the geometry of isosurfaces, were proposed. Klein *et al.* [38] proposed a method explicitly extracting the isosurfaces. The extracted geometry is directly written to an onboard graphics memory object allowing for direct rendering without further bus transfers. The extracted geometry can be manipulated by shader programs and read back to the application for further processing. In [37], Kipfer and Westermann proposed an isosurface extractor that reduces both numerical computations and memory access operations. Given this process, interactive smooth shading and transparent rendering by GPU-based sorting are achieved. However, we have to mention that, most of these methods are based on MT algorithm. In general, MT generates much more triangles than MC. In this paper, we present a GPU-friendly MC algorithm. Moreover, our system can correctly visualize multiple translucent isosurfaces, without sorting.

5.3 Traditional Method

In order to describe the proposed method precisely, we will first introduce some basic concepts of isosurface extraction and associated notations. In our paper, extraction includes two parts: reconstruction and rendering. Reconstruction means computing the surface geometry from the volume data. Rendering means displaying the current view on the screen based on the given geometry.

5.3.1 Scalar Volume Data

For each scalar volume data, there is a pair (V, W) to specify it. V is a finite set of 3D points spanning a domain $\Omega \subset R^3$, in that V $= v_i \in R^3; i = 1, ..., n$. W is a finite set of scalar values, sampled at the points of V, and W $= w_i \in R, i = 1, ..., n$.The correspondence between W and V can be described by a scalar field $f(p)$, when $p = (x, y, z) \in| \Omega$, and $f(V) = W$. A mesh Σ subdivides Ω into polyhedral cells, $\sigma j, j = 1, ..., m$. All vertices of Σ are at the points of $V.\Sigma$ can be made of hexahedra (i.e. cubes), tetrahedra, and etc. For example, in MC algorithm, the mesh Σ is a rectilinear grid. The cells are axis-aligned, and grid spacings along the axes are equal. The constructive cells are cubes. Each vertex can be indexed by a coordinate (i, j, k), and vertex $v_{i,j,k}$ has the value $w_{i,j,k}$.

5.3.2 Isosurface Extraction

Given an isovalue $a \in R$, there are a set of sampled positions Sa within the 3D volume Ω with the same value a, that is $Sa = \{p \in \Omega \mid f(p) = a\}$. Here, S_a is called the isosurface of field f at value a. The isosurface Sa can be approximated by a triangular mesh. The mesh is constructed cell by cell. A cell $a_j \in \Sigma$ has vertices $v_{j1,....,}v_{jh}$, with value $w_{j1,....,}w_{jh}$, where h is the number of vertices for each cell. If $min_{i=i,....,h}w_{ji} \leq a \leq max_{i=1,...,h}w_{ji}$, the cell s j is called active cell at isovalue a. This selection can be completed by labeling. If the value at the vertex exceeds or equals the isovalue, we label the vertex with one. If the value is below the isovalue, we label it with zero. If all vertex values of a cell are 1's or 0's, this cell is not active. An active cell contributes to the approximated isosurface for a patch made for triangles. A non-active cell does not produce triangles. Triangles are obtained by intersection points between the edges of active cells and the isosurface. The edges with intersection are called active edges. The intersections are linear interpolations of two end-points of the active edges. The intersection points are called isosurface vertices. The normal at each isosurface vertex is also estimated by linear interpolation, in order to render the isosurface with smooth shading.

5.3.3 Flow Chart

The isosurface extraction flow chart of program is shown in Figure 5.1. shows the flow chart for isosurface extraction.

1. **Vertex labeling** label each vertex according to the comparison between its value and the given isovalue.
2. **Cell selection** locate all active cells s in the mesh S, given the labels of cell's vertices.
3. **Cell indexing** according to the vertices' labels of the active cell, index the cell in a lookup table, and determine its active edges and how corresponding isosurface vertices must be connected to form triangles .
4. **Normal calculation** for each vertex v of the active cells (all cells or only active cells), computing its corresponding surface normal by its neighboring vertices in the 3D volume space.
5. **Interpolation** for each active edges, computing the 3D coordinates and normal direction of its surface vertex by linear interpolation..
6. **Rendering** given the extracted geometry (a set of triangles), render the surface and display it on the screen.

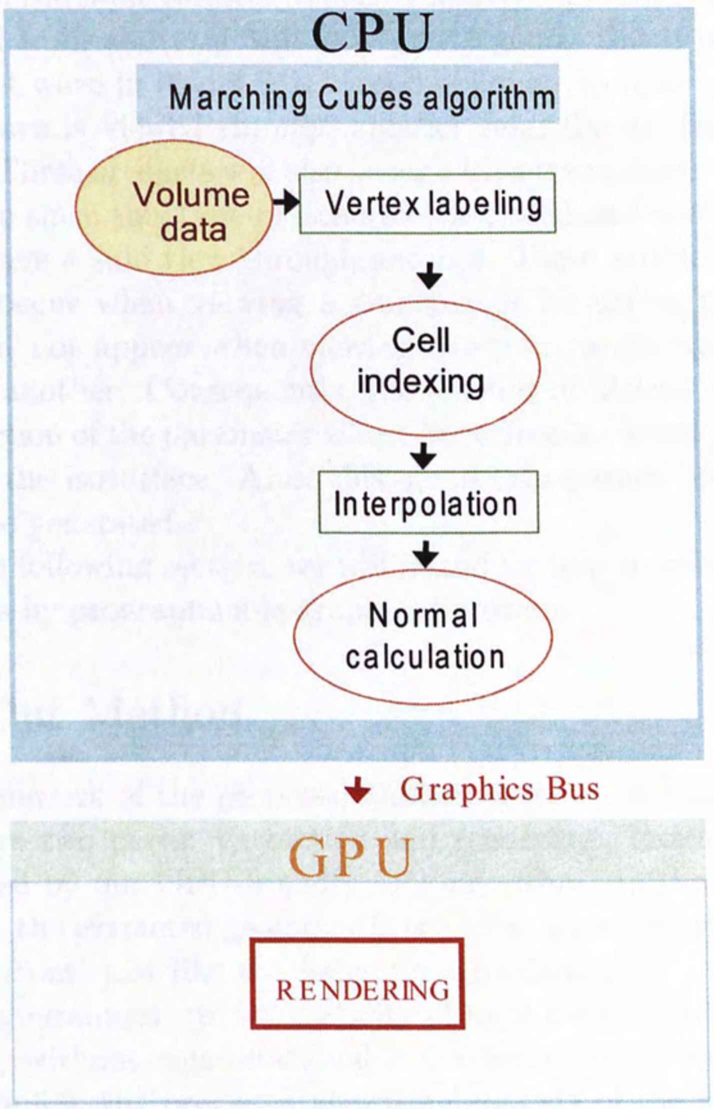


Figure 5.1: CPU Flow Chart.

5.3.4 Transparent Isosurfaces

We use alpha and opengl function glBlend to control transparent surface to support hardware rendering for OpenGL. To render the transparent surface and display it on the screen, we will see artifacts from the rendering as Figure 5.2, processing when a portion of a transparent surface is viewed through another portion of the same surface. When a relatively smooth isosurface is viewed from above, it will look pretty good. But if it has a significant wave in it and it is viewed obliquely so that one side of the wave is viewed through another side, the artifacts will appear. These artifacts will also occur with a transparent blobby isosurface since this type of isosurface is closed and will almost always have a side view through another. These artifacts seem to only occur when viewing a transparent isosurface through itself, but not appear when viewing one transparent isosurface through another. Consequently, the amount of blotchiness will be a function of the parameter whose isosurface is viewed and the value of the isosurface. After this, good transparent isosurface should be generated.

In the following section, we will introduce how to solve these problems by programmable graphics hardware.

5.4 Our Method

The framework of the proposed method is shown in Figure 5.3. There are two parts: extraction and rendering. Extraction is completed by our GPU-friendly MC algorithm. In the rendering part, the extracted geometry is rendered layer by layer, from back to front, just like the painter's algorithm. Our rendering process guarantees correct visibility of multiple translucent isosurfaces, without computational expensive sorting. As shown in Figure 5.3, the proposed algorithm consists of the following

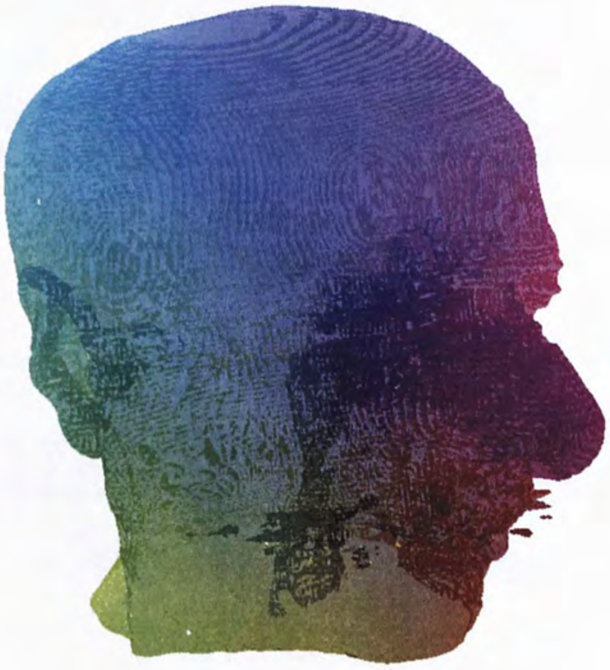


Figure 5.2: Artifacts from the rendering.

main steps:

1. **Cell selection** Active cells are determined in the 3D volume data..
2. **Vertex labeling** Each vertex of the active cell is marked by comparison with the given isovalue α.
3. **Cell indexing** According to the labels, the active cell is indexed in lookup tables, the active edges are determined and how corresponding isosurface vertices must be connected to form triangles is decided .
4. **Normal calculation** Vertex normal is computed by neighboring vertices in this step.
5. **Interpolation** For each active edges, it computes the position and gradient of the corresponding isosurface vertex

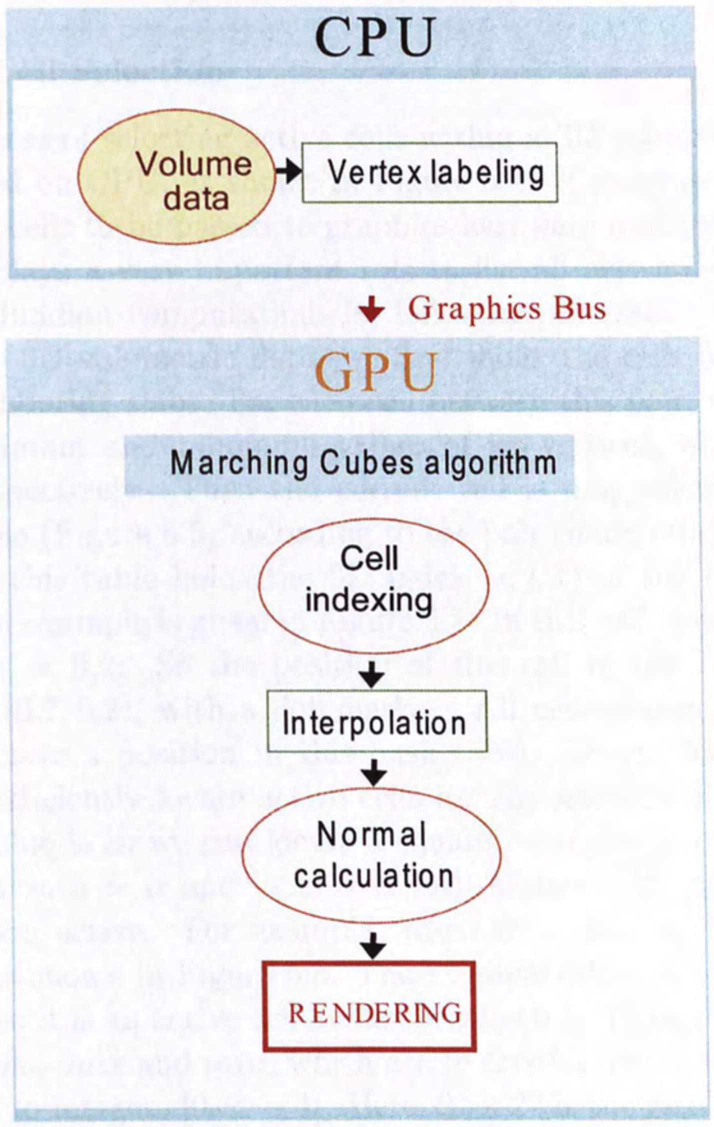


Figure 5.3: GPU Flow Chart.

by linear interpolation. .

6. **Rendering** Given the extracted geometry, it draws triangles from back to front, with the painter's algorithm.

In the proposed method, the vertex labeling is executed on CPU. Cell selection and indexing, normal calculation, interpolation, and rendering are completed by GPU.

5.4.1 Cell Selection

The process of selecting active cells within a 3D volume is implemented on CPU, as shown in Figure 5.4. It generates a set of active cells to be passed to graphics hardware pipeline. This process plays a very important role to list all active cells and avoid redundant computations for following processes.

In the 3D volumetric data, we first index the cells between two neighboring slabs. For each cell between this pair, we find the maximum and minimum values of its vertices, max and min, respectively. Then the current cell is mapped to a 2D hash table (Figure 5.5) according to the pair (max, min). Each entry in this table holds the 3D index (i, j, k) of the mapped cube. An example is given in Figure 5.5. In this cell, $max = 0.7$ and $min = 0.2$. So the position of this cell in the 2D hash table is $(0.7, 0.2)$, with a dot marker. All cells within the 3D volume have a position in this hash table. Given this table, we can efficiently locate active cells for any isovalue α. When the isovalue is α, we can locate a square, which is specified by two lines $min = \alpha$ and $max = \alpha$. All hashed cells inside the square are active. For example, when $\alpha = 0.5$, we locate a square as shown in Figure 5.5. The example cell locates in this square, so it is an active cell at the isovalue 0.5. To manage this hash table, max and min, which are in floating point $[0, 1]$, are mapped to integers $[0, 2^n - 1]$. Here, $2^n \times 2^n$ is the resolution of this table. The larger the table is, the higher precision we can

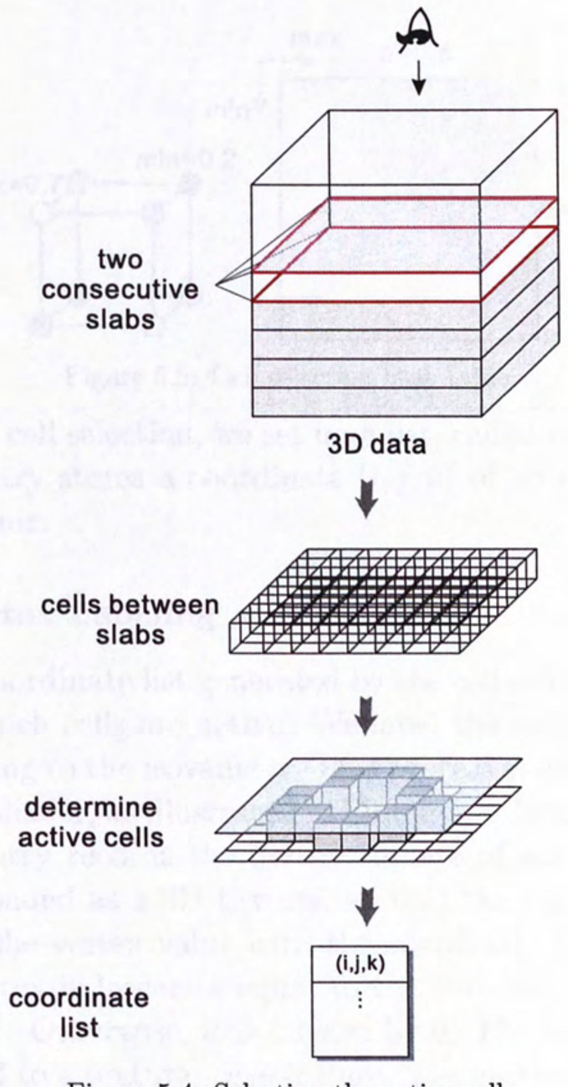


Figure 5.4: Selecting the active cells.

achieve. Since, *max* and *min* are two coordinates of this table and *max* is always larger or equal to *min*, so only half of the table is occupied and only entries in the upper triangular region may have records.

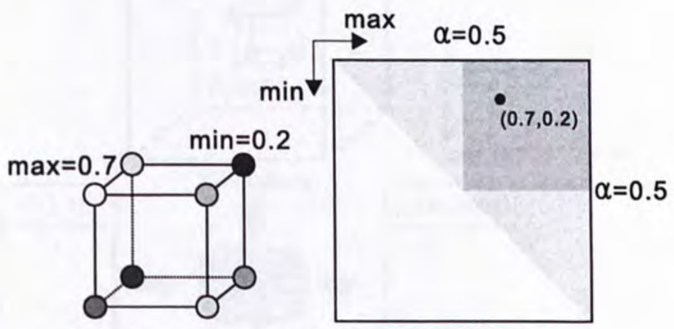


Figure 5.5: Cell selection hash table.

After the cell selection, we set up a list, called *coordinate list*, and each entry stores a coordinate (i, j, k) of an active cell, in the 3D volume.

5.4.2 Vertex Labeling

Given the coordinate list generated by the cell selection process, we know which cells are active. We label the vertices of active cells according to the isovalue α. This process is implemented in a fragment shader, as illustrated in Figure 5.6. In the coordinate list, each entry records the 3D coordinate of active cell. The volume is loaded as a 3D texture, so that the fragment shader can access the vertex value with the coordinate (i, j, k). if the value of vertex is larger or equal to the isovalue, this vertex is labeled by 1. Otherwise, it is labeled by 0. The labeling results are rendered to a texture, *case texture*. The entry of this texture has four components. The first three store (i, j, k), and the last component contains the 8-bit lookup index. The format of the case index is shown in Figure 5.7. Each bit contains the label (1 or 0) of corresponding vertex. For example in Figure 5.6, the cell with the coordinate (i, j, k) is labeled by the isovalue and the case index is 10010000.

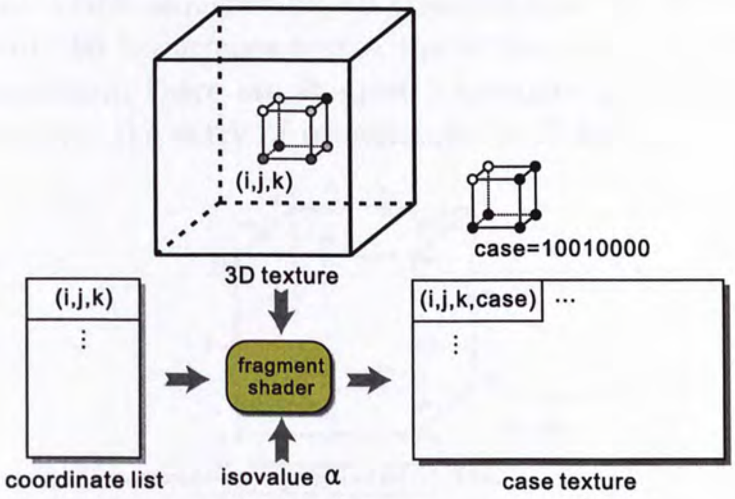


Figure 5.6: Labeling vertices of active cells.

Given the case texture, we can march active cells in the original lookup tables of MC algorithm. Our approach exploits OpenGL framebuffer object [62] to store the rendered texture and/or transfer it from GPU framebuffer to CPU memory.

5.4.3 Cell Indexing

The process of cell indexing is completed by CPU. In MC algorithm, there are two lookup tables: edgetable and triangletable. By indexing an active cell in these two tables, we can determine its active edges and how corresponding isosurface vertices are connected to form triangles. The formats of case index, entries of edgetable and triangletable are shown in Figure 5.7. An 8-bit case index is formed as each bit corresponds to a vertex, from v_7 to v_0. By looking up the edgetable, we get a 12-bit number, each bit corresponding to an edge, from e_{11} to e_0. Value 1 means that the corresponding edge is intersected and active. Value 0 means no intersection. The triangletable involves forming the correct facets from the positions that the isosurfaces intersect the active edges. The lookup table utilizes the same index and gets the vertex sequence for all triangles that are necessary to represent the isosurfaces within the active cell. As defined in MC algorithm, there are at most 5 triangles generated in one cube, so that the entry of triangletable is 15 bytes.

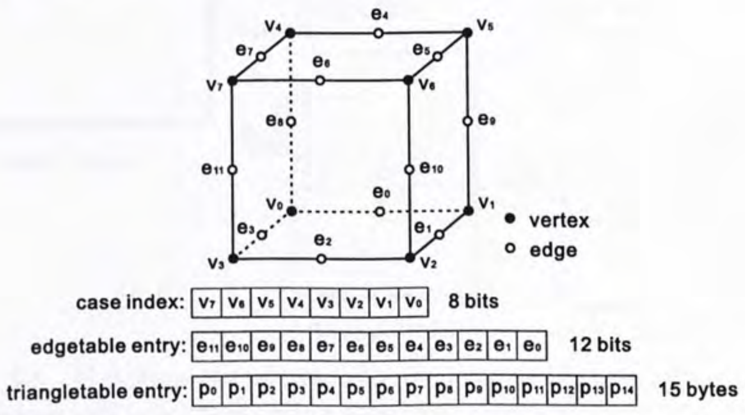

V4
e4
V5
e7
e6
e5
V7
V6
e8
e9
e11
e10
e0
V0
V1
e3
e2
e1
V3
V2
● vertex
○ edge
case index: V7 V6 V5 V4 V3 V2 V1 V0 8 bits
edgetable entry: e11 e10 e9 e8 e7 e6 e5 e4 e3 e2 e1 e0 12 bits
triangletable entry: p0 p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 p12 p13 p14 15 bytes

Figure 5.7: Formats of case index, entries of edgetable and tiangletable.

Given the case texture rendered by the labeling process, for each active cell, we can look up the active edges and corresponding triangle vertices in this cell, as shown in Figure 5.8. All active edges are stored in the edge list. Each entry in the list has three components (x, y, n), where (x, y) is the coordinate of current active cell in the case texture and n is the sequence number of the edge in this cell. The vertex sequence is stored in the vertex list. The entry of this list only has one component m, which is an index of active edge in the edge list. The active edge with the index m is the edge containing the current isosurface vertex. Here the vertex index is interchangeable to active edge index, because each active edge has only one isosurface vertex. An example is illustrated in Figure 5.8. An active cell is with case 10010000. By looking up edgetable, we get 100101010000. This number means edges 4,6,8,11 are active. These four edges are stored in order in the edge list. By looking up triangletable,

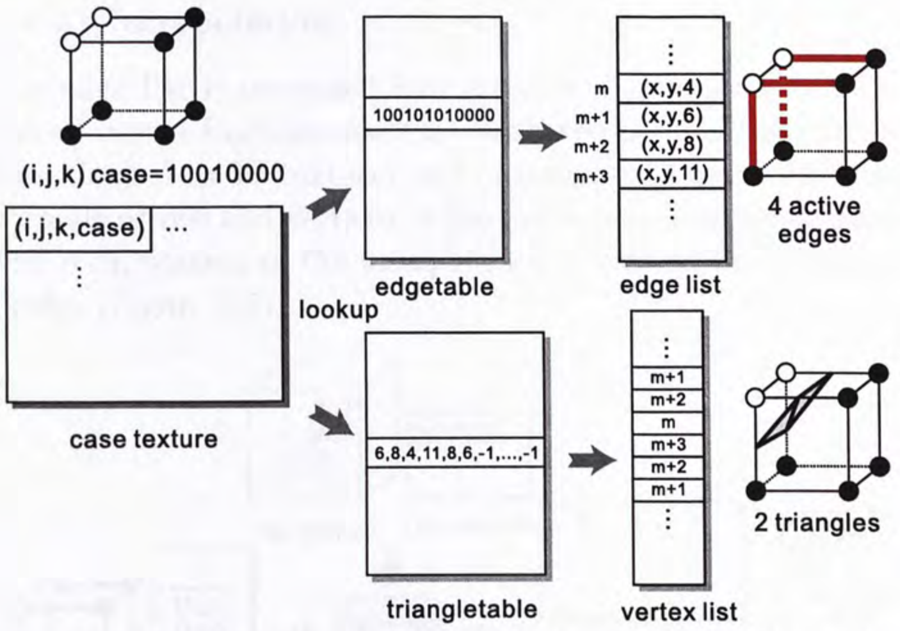


Figure 5.8: Marching the active cells by looking up two tables to obtain active edges and isosurface vertex sequence.

we get $\{6, 8, 4, 11, 8, 6, -1, \cdots, -1\}$. That means there are 2 triangles in this case. "-1" means null. The vertices sequences of two triangles are $\{6, 8, 4\}$ and $\{11, 8, 6\}$, respectively. Actually, there are 15 numbers in this entry. After vertex 6, they are all -1. We will not render vertices with the null value -1. After this process, the edge list is passed to the interpolation process. The vertex list is prepared for rendering process.

This CPU-based process is crucial in selecting active edges in each active cell and list the isosurface vertices one by one. With this process, the performance of interpolation and rendering is significantly improved. In fact, we have tested to move the marching process from CPU to GPU. However, due to computation overhead. The GPU version is slower than that of the current CPU version.

5.4.4 Interpolation

The edge list is processed into a series of fragments and then passed into a fragment shader. With the access of 3D volume data (loaded as 3D texture) and case texture, the positions and normals of two end vertices of the active edge can be achieved. The computation of the interpolation is carried by a fragment shader (Figure 5.9).

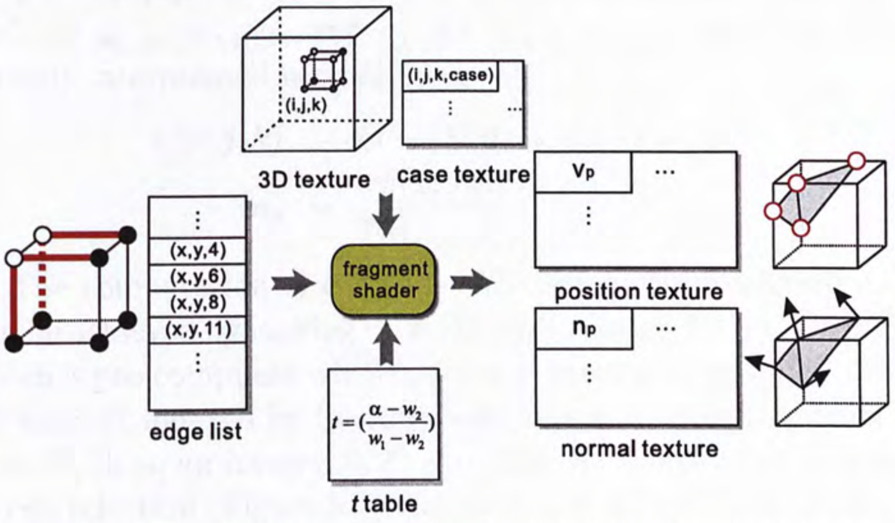


Figure 5.9: Interpolating position and normal of isosurface vertex by a fragment shader.

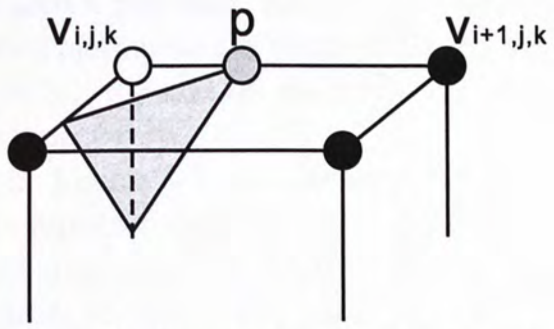


Figure 5.10: Interpolation on each active edge.

As shown in Figure 5.10, there is an intersection point p on the active edge with two end points $v_{i,j,k}$ and $v_{i+1,j,k}$. Given an isovalue α, the position v_p and the normal n_p can be computed by interpolation. The values of $v_{i,j,k}$ and $v_{i+1,j,k}$ are $w_{i,j,k}$ and $w_{i+1,j,k}$, respectively. In the object coordinate, the positions of $v_{i,j,k}$, $v_{i+1,j,k}$, and v_p are (i,j,k), $(i+1,j,k)$, and (x,j,k). The linearly interpolated value x between i and $(i+1)$ is

$$\begin{aligned} x &= (1-t)\cdot i + t\cdot(i+1) \\ &= i+t \\ \text{where } t &= (\frac{\alpha - w_{i,j,k}}{w_{i+1,j,k} - w_{i,j,k}}) \end{aligned}$$

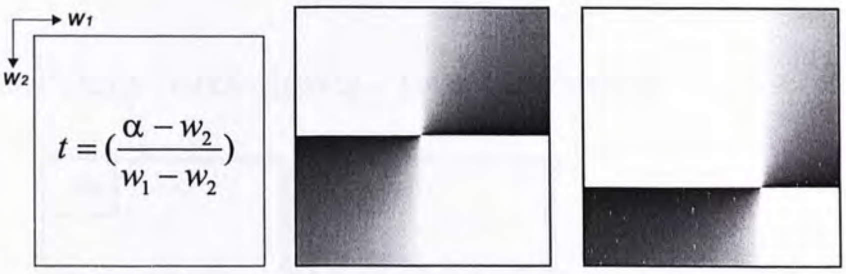
So, in this case $v_p = (i+t, j, k)$.

The computation of normal is a little bit complex. The normals of $v_{i,j,k}$, $v_{i+1,j,k}$, and v_p are $n_{i,j,k}$, $n_{i+1,j,k}$, and n_p. The linearly interpolated normal n_p is

$$\begin{aligned} n(x,j,k) &= (1-t)\cdot n_{i,j,k} + t\cdot n_{i+1,j,k} \\ n_p &= \frac{n(x,j,k)}{\|n(x,j,k)\|} \end{aligned}$$

The computation of t is time consuming. To accelerate it, t can be achieved by looking up a 2D table (Figure 5.11(a) *t table*), which is pre-computed whenever a new isovalue is provided. The 2D table is indexed by (w_1, w_2), and here w is converted from a float $[0,1]$, to an integer $[0, 2^n - 1]$. Similar to the table utilized in cell selection (Figure 5.5), the precision is controlled by n. A larger table will give a higher precision. Two examples $\alpha = 0.5$ and $\alpha = 0.7$ (Figures 5.11(b) and (c)) are given. Since all cells looked up are active, the value of α must be between w_1 and w_2. In this table, two rectangles are valid, while the other regions are useless. Figures 5.11(b) and (c) show the t values when $\alpha = 0.5$ and $\alpha = 0.7$, respectively.

As shown in Figure 5.9, the interpolated positions v_p and normals v_n are rendered into two textures, position texture and normal texture, respectively. These two textures are directly accessed by vertex shader in the following rendering process.



(a) t table (b) $\alpha = 0.5$ (c) $\alpha = 0.7$

Figure 5.11: Interpolation lookup table (a) to accelerate the computation of t. Different isovalues determine different tables, such as (b) $\alpha = 0.5$ and (c) $\alpha = 0.7$.

Before interpolating the normal, the normals at two end points of the active edge are first calculated. Obviously, normals may be repeatedly computed, but this mechanism is suitable for SIMD-based parallel GPU. In the proposed algorithm, we calculate the normals in the same way as in the traditional methods.

5.5 Rendering Translucent Isosurfaces

The vertex list contains the indices of vertices, forming the triangles. The list is prepared as a sequence of vertices, and they are sent to a vertex shader. Since positions and normals are generated in the same order as in edge list, the indices in vertex list can trivially look up the corresponding values in the position and normal texture. In the current GPU, textures can be accessed by vertex shader directly, by `GL_NV_vertex_program3` [61]. The vertex shader renders triangles according to the accessed positions and normals, without transferring the data from CPU to GPU.

We are rendering translucent isosurfaces,however traditional depth-buffering cannot correctly resolve visibility. Although depth sorting and BSP-visibility sorting can handle the visi-

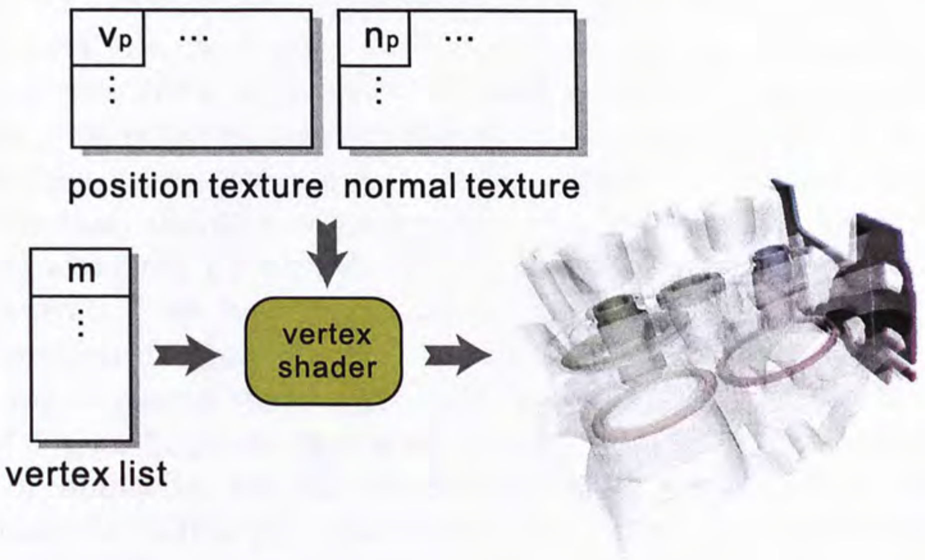


Figure 5.12: Given the vertex list, rendering the isosurfaces triangle by triangle.

bility, they are relatively computational expensive. We use a painter's algorithm that draws the triangles from back to front. To do so, we need to arrange all triangles in a layer data structure. This can be naturally done as we select active cells in a layer by layer fashion. Furthermore, the order of cells specifies the order of extracted triangles. Inspired by [17],

we use Object-Aligned Slicing algorithm to render multiple transparent isosurfaces. In this method, the isosurface components must be sorted in a back-to-front or front-to-back order according to depth from the view-point or viewing plane. We employ a fast yet simple back-to-front sorting method. It essentially exploits a loophole for depth-sorting: the isosurface components in a scene do not have to be truly sorted according to depth from viewing plane, as long as the rendering algorithm guarantees that no isosurface component is drawn after another isosurface component which occludes it.

Therefore three stacks of slices isosurfaces must be stored, with each stack of slices isosurfaces aligned to one of the major axes. Then, the stack with slices isosurfaces most parallel to the screen plane is chosen for rendering the volume.

We run the basic algorithm three times, for x, y, z direction respectively, once along the x direction, once for y direction, and once along z direction. For each direction, we generate a copy of vertex list, position texture, and normal texture, as described in the previous section. For example, in the copy of x direction, triangles are arranged in a layer structure, increasing along the x direction. The copies for y and z are similarly generated. We keep three copies altogether. When the viewing direction changes, our method automatically selects a proper copy to resolve the visibility. For the example in the first row of Figure 5.13, the view point is within the shadowed region. For simplicity, we take the 2D diagram in Figure 5.13 as an example, in this case, the back-to-front direction is the inverse x direction. So, we select the copy of x direction, and draw triangles by inversely visiting the vertex list of this copy. This order guarantees that all triangles are drawn layer by layer, inversely along the x direction. So that the visibility is resolved naturally. For the second example, the back-to-front direction is along the y direction. So we select the copy of y direction, and draw triangles in the order of vertex list of this copy. The translucent isosurfaces are correctly visualized. The second and third columns of Figure 5.13 show intermediate rendering results when 50% and 80% of layers are drawn, respectively. The upper and lower rows show to different viewing orientations.

5.6 Implementation and Results

To evaluate the proposed algorithm, we test it with a variety of 3D volume data, as listed in Table 5.1. All experiments are conducted on a PC equipped with AMD A3800 and GeForce

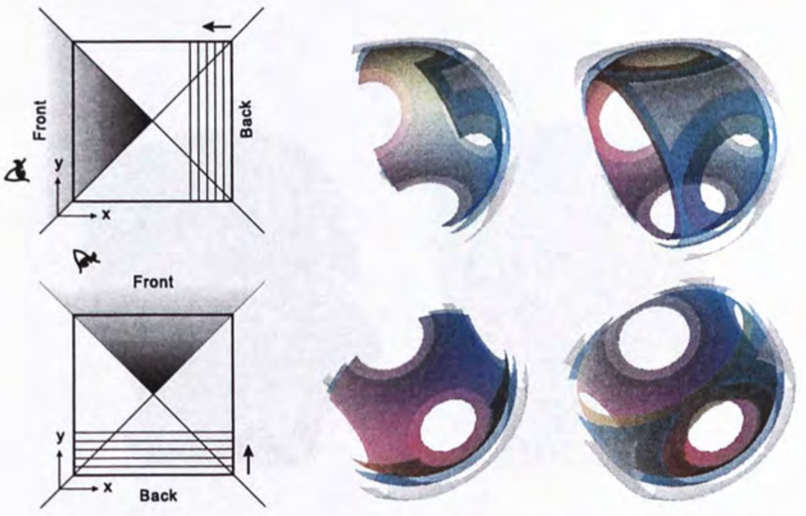


Figure 5.13: The painter's algorithm of rendering multiple translucent isosurfaces.

7800. Statistics are shown in Table 5.1. The statistics includes all processes. In Table 5.1, all isovalues are 0.5. The statistics show that the speed mainly depends on the the number of triangles extracted and rendered. For a high-resolution data set, we can get an interactive isosurfaces extraction and rendering. For low-resolution data, real time performance can be achieved.

Data	Resolution	# Triangles	fps
Sphere	64 × 64 × 64	35,960	27.9
Blood	256 × 256 × 256	132,603	9.9
Head	256 × 256 × 225	651,195	2.2
Inner Ear	128 × 128 × 30	128,291	9.1
Foot	256 × 256 × 256	513,783	2.5
Engine Block	256 × 256 × 128	645,213	2.0
Fuel	64 × 64 × 64	2,242	85.5

Table 5.1: Statistics including all processes.

we will see artifacts from the traditional method rendering as shown in Figure 5.14(left), We have proposed a GPU-friendly al-

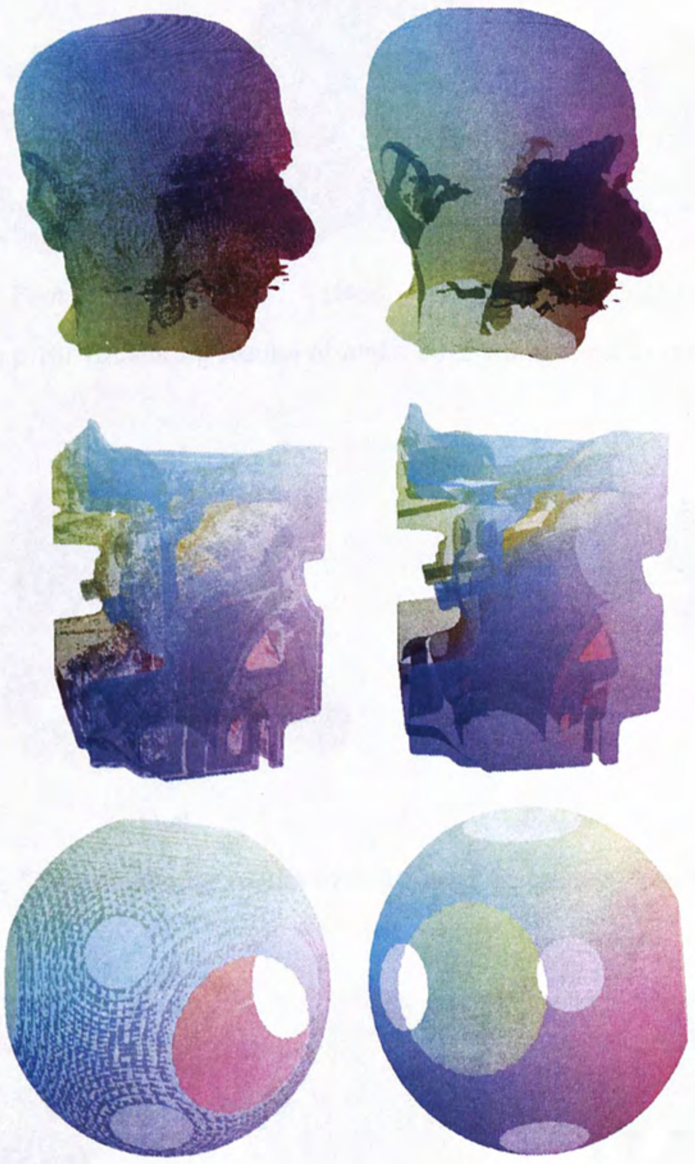


Figure 5.14: Translucent isosurfaces

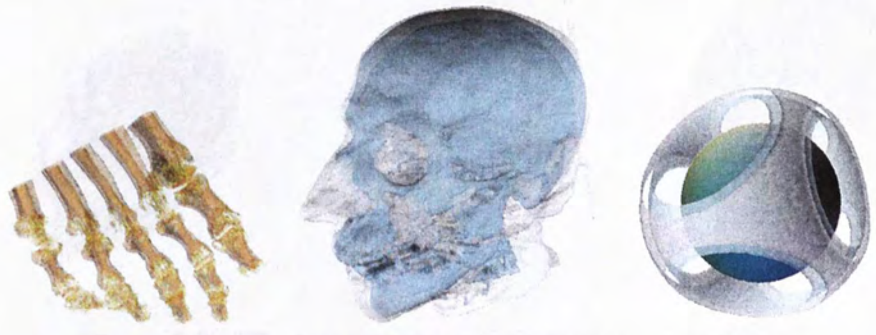


Figure 5.15: Rendering results of multi-layer translucent isosurfaces.

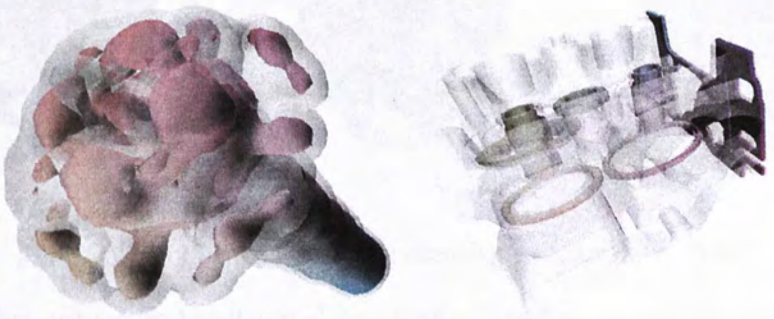


Figure 5.16: Rendering results of multi-layer translucent isosurfaces.

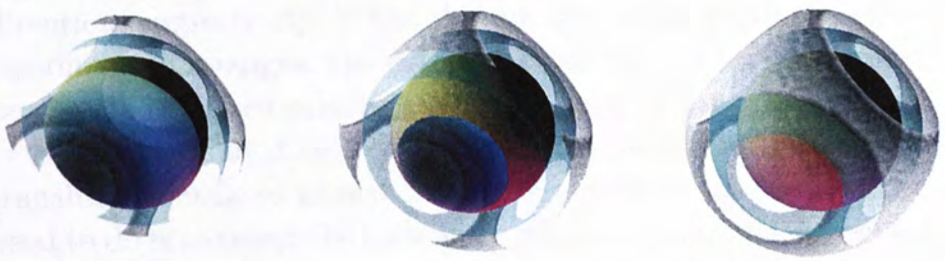


Figure 5.17: Rendered result of "sphere" with 3 isosurfaces.

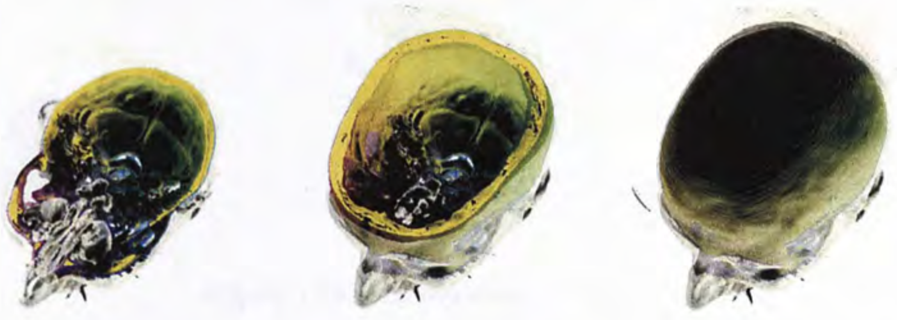


Figure 5.18: The top viewing directions of the data "head".

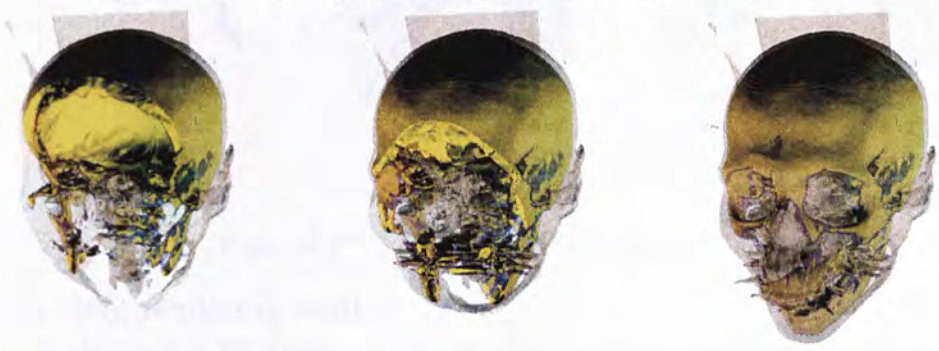


Figure 5.19: The front viewing directions of the data "head"

gorithm,which extracts and renders triangles, in a layer by layer fashion, from back to front, like the painter's algorithm. So we can correctly visualize multiple layers of translucent isosurfaces without performing sorting Figure 5.14(right). We run the basic algorithm three times, along the x direction, y direction, and z direction, respectively. When the isovalue is fixed and the viewing direction changes, the orientation of the slice normal must be changed, by just selecting a proper copy of list and textures.

If the isovalue does not change, the rendering of multiple translucent surfaces is real time in most cases. Because all we need to do is to select the right copy of slices (hence triangles) for display, according the current viewpoint. The rendering timing statistics are shown in Table 5.2. Figure 5.16 shows the corre-

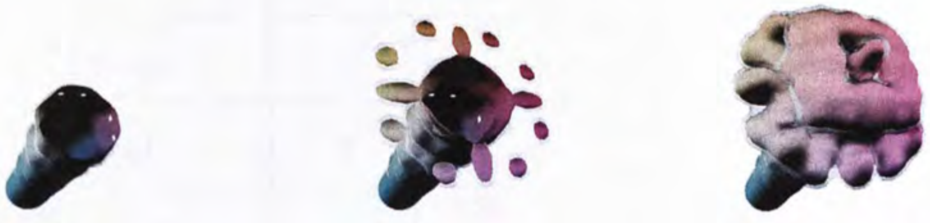


Figure 5.20: The top view of “fuel”.

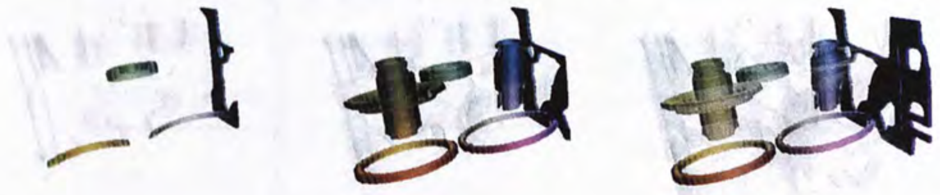


Figure 5.21: The front view of “engine”

sponding rendering results.

In Figure 5.17 ̃Figure 5.21, we show how our system renders the isosurfaces layer by layer, from back to front. Figure 5.17 is the rendered result of “sphere” with 3 isosurfaces. Figure 5.18 and Figure 5.19 are two viewing directions of the data “head”. Figure 5.18 is the top view, and Figure 5.19 is the front view. Figure 5.20 is the top view of “fuel” and Figure 5.21 is the front view of “engine”. Since our system draws triangles, from back to front, we can correctly resolve the visibility of translucent isosurfaces, without sorting.

5.7 Summary

In this chapter, we present a GPU-friendly MC algorithm. The proposed algorithm can rapidly extract and render isosurfaces from high-resolution 3D volume data. Our framework can be trivially modified to implement a wide range of MC variants.

Data	# Layers	Isovalue	# Triangles	fps
Sphere	3	0.44	29,288	170.3
		0.58	30,224	
		0.64	21,272	
Blood	2	0.50	131,423	73.0
		0.73	105,669	
Head	2	0.31	659,202	11.6
		0.44	746,339	
Inner Ear	2	0.38	126,193	70.4
		1.00	140,462	
Foot	2	0.42	708,201	13.1
		0.55	433,468	
Engine Block	2	0.31	564,477	22.9
		0.79	137,736	
Fuel	2	0.02	11,951	455.1
		0.07	8,852	

Table 5.2: Statistics of rendering translucent isosurfaces.

With this framework, we can correctly visualize multiple layers of translucent isosurfaces, without sorting. The proposed framework ensures that the extracted triangles are drawn in a correct order, from back to front, according to the viewing direction. The extracted geometry is stored in GPU memory and ready for post-processing. Given the geometry, many other interesting applications can be developed.

□ **End of chapter.**

Chapter 6

Conclusion

In this thesis, we propose a set of GPU-friendly the volume rendering techniques, including texture-based volume visualization and the famous marching cubes algorithm.

In the technique, we utilize GPU to accelerate the traditional texture-based volume rendering algorithm. A set of slices (2D textures) are indexed from a 3D volume. These slices are aligned with object space. During the rendering process, they are drawn one by one, from back to front. The rendered results show that our method is both efficient and effective.

Then we propose a framework to extract and render iso-surfaces in real time from high-resolution 3D volume data. This framework can be trivially modified to implement a wide range of MC variants. With this technique, we can correctly visualize multiple layers of translucent iso-surfaces without sorting. The proposed framework ensures the extracted triangles are drawn in a correct order, according to the viewing direction.

Within these years, GPU becomes more and more powerful and its programmability also becomes more and more flexible. We can implement more complex algorithms on GPU and lighten the burden of CPU.

□ **End of chapter.**

Bibliography

[1] M. A. O. A. Sud and D. M. Difi. Fast 3d distance field computation using graphics hardware. In *In Proceedings of Eurographics*, pages 117–124, 2004.

[2] J. C. Anderson, J. C. Bennett, and K. I. Joy. Marching diamonds for unstructured meshes. In *Proc. IEEE Visualization*, pages 423–429, 2005.

[3] D. C. Banks, S. A. Linton, and P. K. Stockmeyer. Counting cases in substitope algorithms. *IEEE Trans. Vis. Comput. Graph.*, 10(4):371–384, 2004.

[4] F. Banterle and R. Giacobazzi. A fast implementation of the octagon abstract domain on graphics hardware. In *Proceeding of The 14th International Static Analysis Symposium (SAS)*, pages 315–332, 2007.

[5] M. Bertram. Volume refinement fairing isosurfaces. In *Proc. IEEE Visualization*, pages 449–456, 2004.

[6] A. P. D. Binotto, J. L. D. Comba, and C. M. D. Freitas. Real-time volume rendering of time-varying data using a fragment-shader compression approach. In *Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 69–76, 2003.

[7] S. Boubekeur. Generic mesh refinement on gpu. tamy boubekeur and christophe schlick. In *Proceedings of Graphics Hardware*, pages 99–104, 2006.

[8] B. Cabral, N. Cam, and J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pages 91–98, New York, NY, USA, 1994. ACM.

[9] C. R. J. Charles D. Hansen. The visualization handbook. In *The Visualization Handbook*, pages 888–898, America, 2005. ACADEMIC PRESS.

[10] E. V. Chernyaev. Marching cubes 33: Construction of topologically correct isosurfaces. Technical Report CN/95-17, CERN, 1995.

[11] J. Chhugani, S. Vishwanath, J. Cohen, and S. Kumar. ISOSLIDER: A system for interactive exploration of isosurfaces. In *Proc. of the symposium on Data visualisation*, pages 259–266, 2003.

[12] T. J. Cullip and U. Neumann. Accelerating volume reconstruction with 3d texture hardware. Technical report, Chapel Hill, NC, USA, 1994.

[13] D. A. C.Y. Sun and A. E. Abbadi. Hardware acceleration for spatial selections and joins. In *In Proceedings of SIGMOD*, 2003.

[14] J.-Y. P. S. B. M. G. D.Gdeke, R.Strzodka and S. Turek. Exploring weak scalability for fem calculations on a gpu-enhanced cluster. In *In Proceedings of Parallel Computing*, 2007.

[15] A. Doi and A. Koide. An efficient method of triangulating equivalued surfaces by using tetrahedral cells. *IEICE Trans. Communication, Elec. Info. Syst*, E74(1):214–224, 1991.

[16] K. Engel and T. Ertl. Interactive high-quality volume rendering with flexible consumer graphics hardware. In *Proceedings of EUROGRAPHICS 2002*, 2002.

[17] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 9–16, 2001.

[18] A. K. F. Qiu Z. Fan and S. Yoakum-Stover. Gpu cluster for high performance computing. In *In Proceedings of the ACM/IEEE Super Computing*, 2004.

[19] R. Fernando. Programming techniques, tips and tricks for real-time graphics. In *Addison Wesley Professional, GPU Gems*, 2004.

[20] R. Fernando and M. J. Kilgard. The definitive guide to programmable real-time graphics. In *Addison Wesley Professional, Cg Tutorial*, 2003.

[21] T. Gerstner and R. Pajarola. Topology preserving and controlled topology simplifying multiresolution isosurface extraction. In *Proc. IEEE Visualization*, pages 259–266, 2000.

[22] N. K. Govindaraju, D. Knott, N. Jain, I. Kabul, R. Tamstorf, R. Gayle, M. C. Lin, and D. Manocha. Interactive collision detection between deformable models using chromatic decomposition. *Proc. of SIGGRAPH*, 24(3):991–999, 2005.

[23] B. F. Gregorski, M. A. Duchaineau, P. Lindstrom, V. Pascucci, and K. I. Joy. Interactive view-dependent rendering of large isosurfaces. In *Proc. IEEE Visualization*, pages 475–484, 2002.

[24] A. Guéziec and R. A. Hummel. Exploiting triangulated surface extraction using tetrahedral decomposition. *IEEE Trans. Vis. Comput. Graph.*, 1(4):328–342, 1995.

[25] B. Guo. Interval set: A volume rendering technique generalizing isosurface extraction. In *Proc. IEEE Visualization*, pages 3–10, 1995.

[26] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. In *Proc. EUROGRAPHICS*, pages 303–312, 2005.

[27] M. J. Harris, W. V. B. III, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101, 2003.

[28] E. G. J. Bolz, I. Farmer and P. Schroer. Sparse matrix solvers on the gpu : Conjugate gradients and multigrid. In *In Proceedings of SIGGRAPH*, 2003.

[29] S. L. Junfeng Ji, Enhua Wu and X. Liu. Dynamic lod on gpu. In *Proceedings of Computer Graphics International*, 2005.

[30] M. L. K. E. Hoff, A. Zaferakis and D. Manocha. Fast and simple 2d geometric proximity queries using graphics hardware. In *In Proceedings of the 2001 Symposium on Interactive 3D Graphics*, 2001.

[31] J. S. K. Fatahalian and P. Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *In Proceedings of Graphics Hardware*, 2004.

[32] A. Kaufman and K. Mueller. *Overview of Volume Rendering.* The Visualization Handbook. Academic Press, Inc., 2005.

[33] A. E. Kaufman. 3d volume visualization. In *Advances in Computer Graphics*, 1990.

[34] A. E. Kaufman. Rendering, visualization and rasterization hardware. In *Eurographics*, 1993.

[35] T. Kim and M. Lin. Visual simulation of ice crystal growth. In *In proceedings of ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2003.

[36] P. Kipfer, M. Segal, and R. Westermann. UberFlow: A GPU-based particle engine. In *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, 2004.

[37] P. Kipfer and R. Westermann. GPU construction and transparent rendering of iso-surfaces. In *Proc. Vision, Modeling and Visualization*, pages 241–248, 2005.

[38] T. Klein, S. Stegmaier, and T. Ertl. Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. In *Proc. of Pacific Graphics*, pages 186–195, 2004.

[39] K. G. H. C. Kniss, J. Interactive volume rendering using multidimensional transfer functions and direct manipulation widgets. In *Visualization*, 2001.

[40] A. Kolb, L. Latta, and C. Rezk-Salama. Hardware-based simulation and collision detection for large particle systems. In *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 123–131, 2004.

[41] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization*, page 38, Washington, DC, USA, 2003. IEEE Computer Society.

[42] E. S. Larsen and D. K. McAllister. Fast matrix multiplies using graphics hardware. In *In Proceedings of Supercomputing*, 2001.

[43] A. Lopes and K. Brodlie. Improving the robustness and accuracy of the marching cubes algorithm for isosurfacing. *IEEE Trans. Vis. Comput. Graph.*, 9(1):16–29, 2003.

[44] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics (Proc. of SIGGRAPH)*, 21:163–169, 1987.

[45] T. S. M. J. Harris, W. V. Baxter III and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *In Proceedings of Graphics Hardware*, 2003.

[46] Mason Woo, Jackie Neider, OpenGL Architecture Review Board, Tom Davis, Dave Shreiner. Opengl. Addison-Wesley, 2001.

[47] K. Moreland and E. Angel. The fft on a gpu. In *In Proceedings of Graphics Hardware*, 2003.

[48] K. Moreland and E. Angel. The FFT on a GPU. In *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, 2003.

[49] D. A. N. Bandi, C.Y. Sun and A. E. Abbadi. Hardware acceleration in commercial databases: A case study of spatial operations. In *In Proceedings of VLDB*, 2004.

[50] G. L. D. L. N. Goodnight, C. Woolley and G. Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. In *In Proceedings of Graphics Hardware*, 2003.

[51] M. C. L. N. Govindaraju, S. Redon and D. M. Cullide. Interactive collision detection between complex models in large environments using graphics hardware. In *In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2003.

[52] S. K. N. H. Mustafa, E. Koutsofios and S. Venkatasubramanian. Hardware assisted view dependent map simplification. In *In Proceedings of the 17th Annual Symposium on Computational Geometry*, 2001.

[53] W. W. M. C. L. N. K. Govindaraju, B. Lloyd and D. Manocha. Fast database operations using graphics processors. In *In Proceedings of SIGMOD, June*, 2004.

[54] G. M. Nielson. MC*: Star functions for marching cubes. In *Proc. IEEE Visualization*, pages 59–66, 2003.

[55] G. M. Nielson. On marching cubes. *IEEE Trans. Vis. Comput. Graph.*, 9(3):283–297, 2003.

[56] G. M. Nielson. Dual marching cubes. In *Proc. IEEE Visualization*, pages 489–496, 2004.

[57] G. M. Nielson and B. Hamann. The asymptotic decider: Removing the ambiguity in marching cubes. In *Proc. IEEE Visualization*, pages 83–91, 1991.

[58] G. M. Nielson, A. Huang, and S. Sylvester. Approximating normals for marching cubes applied to locally supported isosurfaces. In *Proc. IEEE Visualization*, pages 459–466, 2002.

[59] O. Nilsson, D. E. Breen, and K. Museth. Surface reconstruction via contour metamorphosis: An eulerian approach with lagrangian particle tracking. In *Proc. IEEE Visualization*, pages 407–414, 2005.

[60] nVidia Coorperation. GL_EXT_framebuffer_object. http://www.nvidia.com/dev_content/nvopenglspecs/ GL_EXT_framebuffer_object.txt.

[61] nVidia Coorperation. GL_NV_vertex_program3. http://www.nvidia.com/dev_content/nvopenglspecs/ GL_NV_vertex_program3.txt.

[62] nVidia Coorperation. The opengl framebuffer object extension, 2005.

[63] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *EUROGRAPHICS, State of the Art Report*, pages 21–51, 2005.

[64] V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. In *Joint EUROGRAPHICS/IEEE TCVG Symposium on Visualization*, pages 293–300, 2004.

[65] T. J. Purcell, C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 41–50, 2003.

[66] Randi J. Rost, John M. Kessenich, Barthold Lichtenbelt. Opengl Shading Language . Addison-Wesley, 2006.

[67] F. Reck, C. Dachsbacher, R. Grosso, G. Greiner, and M. Stamminger. Realtime isosurface extraction with graph-

ics hardware. In *EUROGRAPHICS Short Presentations*, 2004.

[68] N. H. M. S. Krishnan and S. Venkatasubramanian. Hardware assisted computation of depth contours. In *In Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, 2002.

[69] Web. The Marching Cubes Algorithm. http://www.exaflop.org/docs/marchcubes/index.html.

[70] D. Weiskopf, T. Schafhitzel, and T. Ertl. GPU-based nonlinear ray tracing. In *Proc. EUROGRAPHICS*, pages 625–633, 2004.

[71] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proc. the 25th annual conference on Computer graphics and interactive techniques*, pages 169–177, 1998.

[72] J. Wilhelms and A. V. Gelder. Topological considerations in isosurface generation extended abstract. In *VVS '90: Proceedings of the workshop on Volume visualization*, pages 79–86, New York, NY, USA, 1990. ACM.

[73] J. Wilhelms and A. V. Gelder. Octrees for faster isosurface generation. *ACM Trans. Graph.*, 11(3):201–227, 1992.

[74] H. Xie, K. T. McDonnell, and H. Qin. Surface reconstruction of noisy and defective data sets. In *Proc. IEEE Visualization*, pages 259–266, 2004.

[75] X. L. Y.Q. Liu and E. Wu. Real-time 3d fluid simulation on gpu with complex obstacles. In *In Proceedings of Pacific Graphics*, 2005.