REAL-TIME SOLID VOXELIZATION USING MULTI-CORE PIPELINING

by Duoduo Liao

M.S. in Computer Science, August 2001, Purdue University

A Dissertation submitted to

the Faculty of
The School of Engineering and Applied Science
of The George Washington University
in partial satisfaction of the requirements
for the degree of Doctor of Philosophy

May 17, 2009

Dissertation directed by

Simon Y. Berkovich
Professor of Computer Science

The School of Engineering and Applied Science of The George Washington University certifies that Duoduo Liao has passed the Final Examination for the degree of Doctor of Philosophy as of February 26th, 2009. This is the final and approved form of the dissertation.

REAL-TIME SOLID VOXELIZATION USING MULTI-CORE PIPELINING

Duoduo Liao

Dissertation Research Committee:

Simon Y. Berkovich, Professor of Computer Science, Dissertation Director

Abdou Youssef, Professor of Computer Science, Committee Member

Rhys Price Jones, Professor of Computer Science, Committee Member

Murry H. Loew, Professor of Electric and Computer Engineering, Committee Member

Shiaofen Fang, Associate Professor of Computer Science at Indiana University Purdue University Indianapolis, Committee Member

© Copyright 2009 by Duoduo Liao

All Rights Reserved

DEDICATION

To my family

Acknowledgements

I greatly appreciate my advisor, Professor Simon Y. Berkovich for his supervision, advice, and guidance, especially his giving me inspiration to head in the most challenging and promising direction. He also provided me unflinching encouragement and support in various ways. It has been a real pleasure and privilege working with him.

I gratefully acknowledge my dissertation committee members for their thoughtful advice, encouraging words, and time and attention during busy semesters. It is a pleasure to pay tribute to Professor Abdou Youssef for his insightful advice and support since the early stage of my doctoral study. I am very thankful to Professor Rhys Price Jones for his helpful comments and advice in my dissertation writing. Many thanks go to Professor Murry H. Loew for his insights and furthermore, using his precious time to collaborate with me on the paper. I am very grateful to Professor Shiaofen Fang, whose valuable advice has triggered and nourished my research in volume graphics.

Sincere appreciation is extended to Dr. Timothy Tong, for his support and encouragement. I also would like to thank Professor James K. Hahn for his advice on my initial doctoral program and for the dissertation advisor recommendation.

Finally, words fail me to express my deepest appreciation to my family for their endless support, patience, and love. My parents continue to be my role models for living life with passion, creativity, and hard work. I owe my sincere gratitude to my husband and dear baby for everything. All my family's dedication and love keep me persistently devoted to my research throughout this long journey.

Abstract

REAL-TIME SOLID VOXELIZATION USING MULTI-CORE PIPELINING

This dissertation not only proposes a new approach for real-time solid voxelization by Graphics Processor Unit (GPU) acceleration, but also proposes a novel multi-core pipelined parallel architecture to further improve total performance. Especially, the new generic parallel architecture can also be successfully applied to other multi-core/many-core architectures or heterogeneous systems. The most important advantage of this new architecture is that most existing single-processor software programs can be directly used with nearly no change, unlike conventional parallel programming which has to take into account data distribution, load balancing, and scheduling. The algorithms and architectures introduced in this dissertation not only improve performance and quality, but also are convenient to implement and integrate into many applications. The experimental results, performance analysis, comparison, and case studies demonstrate the effectiveness, flexibility, and diversity of developed approaches and architecture systems.

The GPU accelerated real-time solid voxelization approach utilizes an innovative dynamic slice function mechanism and masking techniques to significantly improve solid voxelization speed in real time as well as create various multi-valued solid volumetric models. In particular, by dynamically applying different slice functions, any surface-closed geometric model can be voxelized into a solid volumetric representation with various multi-valued interior features, such as rainbow, marble, wood, or translucent jade. The algorithms are easy to implement and convenient to integrate into diverse

applications, such as volume modeling, collision detection, volume rendering, medical simulation, and computer art.

An effective scheme for real-time parallel solid voxelization is presented using multi-core pipelining. A novel generic multi-core pipelined parallel architecture is first proposed based on program slicing by forced interrupts and a distinct memory/cache dynamic management technology. A new parallel solid voxelization system is discussed based on this novel architecture extended to multi-core GPUs. These architectures provide simple and effective solutions to on-the-fly computations by transferring the operating states from processor to processor. They require practically the same software as is currently used on a single-CPU system (i.e. a sequential computer), unlike conventional parallel programming costing extra time and effort. The parallel solid voxelization experimental results, performance analysis, and comparison with existing multi-core GPUs show the effectiveness of the new architectures.

At last, the two important volume graphics applications based on the real-time GPU-accelerated solid voxelization, volumetric Constructive Solid Geometry (CSG) modeling and volumetric collision detection, are further studied in detail. First, a new algorithm of fast volumetric CSG modeling for volume scene by scene computation is proposed. It is capable of carrying out interactive modeling of heterogeneous volume scene. Second, a simple and fast collision detection method is presented as one of the successful applications using this core voxelization approach. It improves the collision detection performance as well as detection accuracy based on voxel level.

Table of Contents

Dedication ..... **iv**

Acknowledgements ..... **v**

Abstract ..... **vi**

Table of Contents ..... **viii**

List of Figures ..... **xiii**

List of Tables ..... **xvii**

List of Acronyms ..... **xviii**

1 Introduction **1**

1.1 Problem Statement 1

1.1.1 Significance of Volumetric Representation 1

1.1.2 Difficulty of Volumetric Data Acquisition 3

1.1.3 Difficulty of Parallel programming 5

1.2 Motivation 5

1.3 Objective 7

1.4 Proposed Solutions 8

1.5 Contributions 10

1.6 Document Organization 12

1.7 Overview of the Proposed Systems 13

2 Background **14**

2.1 Volume Data 14

2.2 Volume Graphics 15

2.3 Computer Graphics Hardware and APIs 16
2.3.1 3D Graphics APIs 16
2.3.2 GPU 16
2.3.3 Graphics Hardware Pipeline 19
2.4 Solid Voxelization 20
2.4.1 Prior Work of Solid Voxelization 21
2.4.2 Classification and Comparison of Existing Solid Voxelization Algorithms 24
2.4.3 Parallel Voxelization 27
2.5 Noise Functions 27
2.6 Constructive Solid Geometry 28
2.7 Volumetric CSG Model 29
2.8 Parallel Computing 31
2.9 Multi-processor Pipeline by Forced Interrupts 31
2.10 Multi-core GPU Architectures 33
2.11 Crossbar-based Technologies 34
2.12 Multi-core GPU Programming 35
3 Real-Time Solid Voxelization by GPU Acceleration 37
3.1 Solid Object 38
3.1.1 Solid Volumetric Object 38
3.1.2 Solid Geometric Object 39
3.1.3 Definition of Solid Voxelization 40
3.2 Solid Slice Function 41
3.2.1 Definition of Solid Slice 41

3.2.2 Definition of Solid Slice Function 42
3.2.3 Several Typical Solid Slice Functions 42
3.3 Solid Slice Creation 44
3.3.1 Surface Parity Check and Mask Creation 44
3.3.2 Principle of Solid Slice Creating 48
3.3.3 Algorithm of Solid Slice Creating 49
3.3.4 Advantages of Solid Slice Creating Algorithm 51
3.4 GPU-based Real-time Solid Voxelization 52
3.4.1 Algorithm of Real-time Solid Voxelization 52
3.4.2 Real-time Acceleration Techniques 55
3.5 Experimental Analysis 56
3.5.1 Experimental Results 56
3.5.2 Performance 64
3.5.3 Analysis 71
3.6 Advantages of GPU-based Real-time Solid Voxelization 77
4 A New Multi-Core Pipelined Architecture by Forced Interrupts 79
4.1 Original Multiprocessor Pipeline by Forced Interrupts 80
4.2 New Multi-core Memory and Cache Architecture Based on Crossbar Switch 82
4.3 New Multi-core Pipeline Organization 84
4.4 Timing Diagram 86
4.5 Program Requirements 88
4.6 Performance Analysis 88
4.6.1 Time Complexity 88

4.6.2 Performance 91
4.6.3 Experimental Analysis 92
4.7 Summary and Advantages 99
5 Parallel Solid Voxelization Using Multi-Core Pipelining 102
5.1 A Multi-core Pipelined GPU Architecture 103
5.1.1 The Principle of the New GPU Architecture 103
5.1.2 Comparison with Existing Multi-core GPUs 104
5.2 A Multi-Core Pipelined System for Parallel Solid Voxelization 106
5.2.1 The System Architecture 106
5.2.2 The Parallel Solid Voxelization Algorithm and Comparison 108
5.3 Principle of Parallel Solid Voxelization by Multi-Core Pipeline 111
5.4 Experimental Analysis 113
5.4.1 The Performance 113
5.4.2 Experimental Results and Comparison 114
5.5 Summary and Advantages 118
6 Case Study 1: Volumetric CSG Modeling 119
6.1 CSG Model 120
6.2 Volume Scene Tree 122
6.2.1 Definition of Volume Scene Tree 122
6.2.2 Operator Functions 123
6.3 Volume Scene Computation 125
6.3.1 The Principle of Slice Sweeping 126
6.3.2 Algorithm of Scene Computation 127

6.3.3 Pipeline of Slice Sweeping 129
6.3.4 Slice Computation - Voxelization 131
6.3.5 Time Complexity 133
6.4 VST Pruning 134
6.4.1 Definitions of Operators 134
6.4.2 Pruning cases 135
6.4.3 Pruning algorithm 136
6.5 Experimental Results 137
6.5.1 VCSG Modeling Examples and Time Spent 137
6.5.2 Examples of VCSG Modeling for Other Applications 140
6.6 Chapter Summary 143
7 Case Study 2: Volumetric Collision Detection 144
7.1 Color Coding and White Pixel Check 145
7.2 VST Design for Volumetric Collision Detection 146
7.3 Algorithm of Volumetric Collision Detection 147
7.4 Experimental Results 148
8 Conclusions 152
8.1 Summary 152
8.1.1 Benefits 152
8.1.2 Limitations 155
8.2 Future Work 156
References 158

List of Figures

Figure 1.1: Surface rendering for a dragon bronze 2

Figure 1.2: Examples of voxelization 4

Figure 1.3: Overview of proposed systems..... 13

Figure 2.1: The graphics hardware pipeline 19

Figure 2.2: The graphics pipeline 20

Figure 2.3: An example of VCSG model..... 29

Figure 3.1: A slice of a solid volumetric object..... 39

Figure 3.2: Determination for frontface and backface..... 45

Figure 3.3: Surface parity check and mask creating 45

Figure 3.4: Principle of one solid slice creating 48

Figure 3.5: Algorithm of solid slice creating 50

Figure 3.6: Algorithm of real-time solid voxelization 54

Figure 3.7: Solid voxelization examples of the Dragon geometric model 58

Figure 3.8: Solid voxelization results in 256^3 of the Dragon geometric model 59

Figure 3.9: Solid voxelization examples of the Buddha geometric model 61

Figure 3.10: Solid voxelization results in 256^3 of the Buddha geometric model..... 62

Figure 3.11: Solid voxelization results in 256^3 of the other models 63

Figure 3.12: XOR-based Solid voxelization results in 128^3 of some models 65

Figure 3.13: The statistic graph of solid voxelization time of Dragon_res4 model with different slice size for a fixed slice number 72

Figure 3.14: The statistic graph of solid voxelization time of Dragon models with FUNC_SINGLE_COLOR 74
Figure 3.15: The statistic graph of solid voxelization time of Buddha models with FUNC_SINGLE_COLOR 74
Figure 3.16: The statistic graph of solid voxelization time of all the models with FUNC_SINGLE_COLOR 75
Figure 3.17: Solid voxelization time of the Dragon_res4 model using a constant slice function 75
Figure 3.18: Solid voxelization time of the Dragon_res4 model using different texture slice functions 76
Figure 3.19: Solid voxelization time of the cow model using different texture slice functions 76
Figure 3.20: The ratio of the voxelization speed of the new algorithm to the XOR-based algorithm 77
Figure 4.1: Helicoidal pattern in on-the-fly computations by a microprocessor pipeline 81
Figure 4.2: Crossbar switch based Multi-core memory and cache architecture 83
Figure 4.3 Improved multi-core pipeline organization and data flows 85
Figure 4.4: Timing diagram of a 3-core pipeline system 87
Figure 4.5: The principle of the *N*-core pipeline 89
Figure 4.6 The performance for different Intervals 93
Figure 4.7: The performance for different number of cores 94
Figure 4.8: Maximum performance for different switching time 96
Figure 4.9: The performance for different dataset size with different number of cores ... 97

Figure 4.10: Maximum performance for different intervals and number of cores 98

Figure 5.1: Crossbar switch based Multi-core GPU architecture 104

Figure 5.2: Multi-core pipeline system for parallel solid voxelization..... 107

Figure 5.3: Sequential solid voxelization algorithm for the Multi-core Pipelined Parallel System..... 109

Figure 5.4: Conventional parallel algorithm for solid voxelization..... 110

Figure 5.5: Flow chart of conventional parallel solid voxelization 111

Figure 5.6: The principle of parallel voxelization by multi-core pipeline..... 112

Figure 5.7: 16 slices of dragon_res4 model after voxelization (128x128x16) 115

Figure 5.8: Time spent on each slice voxelization for dragon_res4 model (128x128x16) 115

Figure 5.9: Multi-core pipelined GPU performance with different switching overheads 116

Figure 5.10: Comparison of multi-core pipelined GPUs and existing Multi-core GPUs117

Figure 6.1: A CSG tree example using a CT dataset 121

Figure 6.2: An example of a volume scene tree..... 122

Figure 6.3: Slicing the volume space..... 127

Figure 6.4: The conceptual scene computation pipeline..... 130

Figure 6.5: Special cases for VST..... 133

Figure 6.6: A pruned volume scene tree example..... 134

Figure 6.7: Some VCSG modeling examples..... 138

Figure 6.8: Some VCSG modeling examples for medical application..... 140

Figure 6.9: Some VCSG modeling examples for GIS applications 142

Figure 7.1: Coloring for 3 objects with 1 collision 145

Figure 7.2: A VST for volumetric collision detection 147

Figure 7.3: An example of volumetric collision detection 149

Figure 7.4: An example of volumetric collision detection in virtual surgery application 151

List of Tables

Table 2.4.1: Comparison of several existing typical solid voxelization algorithms 26

Table 3.2.1: Math function descriptions of some typical solid slice functions..... 43

Table 3.3.1: An example of surface parity check 46

Table 3.5.1: Time and speed comparison of solid voxelization of the Dragon models.... 67

Table 3.5.2: Time and speed comparison of solid voxelization of the Happy Buddha models 68

Table 3.5.3: Time and speed comparison of solid voxelization of the other models 69

Table 3.5.4: Time Spent of solid voxelization using color-based slice functions 70

Table 3.5.5: Time spent on solid voxelization using texture-based slice functions 70

Table 4.6.1: The performance for different number of cores with corresponding pipeline lengths 94

Table 5.1.1: Comparison of multi-core GPU architectures 105

Table 6.4.1: Operation definitions 135

Table 6.4.2: Three cases for VST pruning 136

Table 6.5.1: Voxelization time for different methods with different models 139

Table 6.5.2: Time comparison of two solid voxelization algorithms for the Office model 139

List of Acronyms

CAD Computer-aided design

CAM Computer-aided manufacturing

Cg C for graphics (programming language developed by Nvidia)

CGPU CPU and GPU combination

CPU Central processing unit or just processor

CSG Constructive solid geometry

CT Computed tomography

CUDA Compute unified device architecture

DEM Digital elevation model

DSP Digital signal processing

GIS Geographic information system

GPGPU General-purpose graphics processor unit

GPU Graphics processor unit

L Loading

LMA Lightspeed Memory Architecture™

LOS Line of sight

MRI Magnetic resonance imaging

OpenCL Open computing language

OpenGL Open graphics library

P Processing

ROI Region of interest

SIMD Single Instruction Multiple Data

S Switching

U Unloading

UMA Uniform Memory Access

USGS United States Geological Survey

VCD Volumetric collision detection

VCSG Volumetric CSG

VGIS Volumetric GIS

VST Volume scene tree

WPC White pixel check

XOR Exclusive disjunction

1 Introduction

Recently, volumetric data have been used in a wide range of applications including scientific visualization, medical imaging, biomedical education and training, prototyping in Computer-Aided Manufacturing (CAM) or Computer-Aided Design (CAD), 3D printing, visual simulation and virtual reality, and digital characters and environments for the entertainment industry. There is a tremendous demand for a wide variety of volumetric models and the approaches to generate volumetric data. While there has been significant progress in the area of volume rendering for over the past decades, creating and acquiring high-fidelity volumetric models remains a challenging and expensive process. Furthermore, since the size of volumetric data is usually very large, massive volumetric data processing using parallel computing has been among top challenges in volume graphics.

1.1 Problem Statement

1.1.1 Significance of Volumetric Representation

Most of computer graphics imagery is created with surface representations. Popular surface modeling tools, such as Softimage, Autodesk's AutoCAD and Maya, can be utilized to compose beautiful imagery based on surface rendering, such as a dragon bronze, shown in Figure 1.1. But surface representations are unable to capture object properties beyond shape and texture. Specifically, they are not adequate for representing

interior features, underlying structures, or some physics-based operations, such as cracked bricks, eroded metals, or the effects how the inside changes over time. Since interior samples are usually exposed and most objects are amorphous, volumetric representations appear to be a natural choice.

Deformations of surface model can result in overall volume change or introduce self-intersections. It can be difficult to perform simple changes in topology, such as drilling a hole through the model. These operations are best performed on a volumetric representation.

Figure 1.1: Surface rendering for a dragon bronze

Collision detection is very important for interactive applications, such as visual simulation and gaming. However, in traditional collision detection, many complicated mathematical computations have to be employed to detect the collisions for the surface model. These algorithms are very time-consuming due to the complex intersection operations between points, lines, and planes. Since volumetric data are only represented by a set of voxels, the intersection operation is only applied between a voxel and a voxel,

(i.e., a point and a point). Thus the collision detection becomes simple and fast based on volumetric representation.

1.1.2 Difficulty of Volumetric Data Acquisition

One great challenge in volume graphics today comes from how to acquire the volumetric data of 3D objects and how to fast acquire internal data in real time.

Some biomedical scanning devices, such as CT, MRI, con-focal microscopes, or other tomography and/or dissection techniques can acquire some internal features. However, there methods can be slow, expensive, destructive, impractical, or used for certain materials or objects.

Three-dimensional digitizing has emerged as a popular technique for acquiring complex models, such as sculptures or mechanical parts. While such digitizers are useful for acquiring surface shape and appearance properties, they do not capture the internal structure of the geometry, which may be necessary for accurate animation or simulation. Often even the surface texture is lost since the object must be painted matte white for scanning.

Voxelization is another way to acquire the internal volumetric data of geometric objects. It is a process to convert geometric objects from their continuous geometric representation into a set of voxels which best approximates the continuous objects. This is an indispensable stage in volume graphics [Kau91, Kau93].

The earliest voxelization idea, 3D scan-conversion, first proposed by Kauffman in [Kau86], for voxel-based volume, was inspired by the extension of 2D scan-conversion for a pixel-based image. The earliest voxelization research started with geometric objects, such as lines, curves and surfaces. The voxelization for surface-based objects, called

surface voxelization, where merely the surface boundaries of the objects are voxelized into a set of voxels, shown in Figure 1.2 (a). Currently, a large amount of such research still continues. But this kind of voxelization does not provide internal features either.

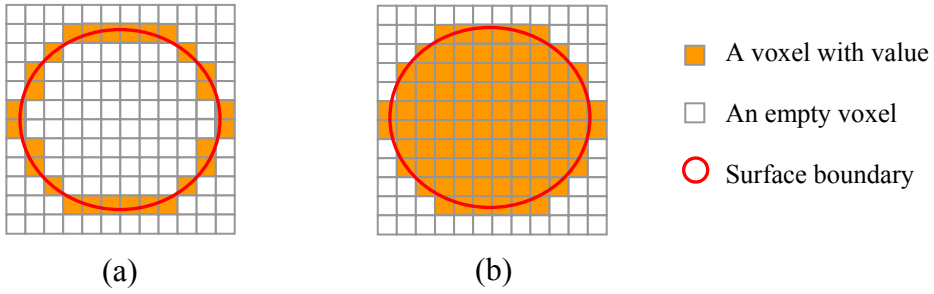


Figure 1.2: Examples of voxelization
(a) Surface voxelization, (b) solid voxelization

Unlike surface-based geometric objects, solid geometric objects are normally represented by their boundary surfaces without explicit interior information. The voxelization of solid geometric objects, called *solid voxelization*, can obtain the interior features of objects as shown in Figure 1.2 (b). However, solid voxelization is much more difficult and time-consuming because this process requires an inside test for each voxel in the volume space.

Generally speaking, less research on solid voxelization has been carried out, although the research of pure software-based solid voxelization started early [Lee82]. With the development of graphics hardware in recent years, some solid voxelization methods based on pure software, hardware acceleration, or software and hardware mixing have been studied [Kar99, Fan00b, Bae00, Lia02, Hau02, Var03, Rui04, Pas04]. Unfortunately, limitations in algorithms, such as low speed, missing voxels, slice dependency, concave problems, binary-only voxels, discontinuous voxelization, non-transparency, etc., still exist. Section 2.4 will summarize these methods and give the detailed comparisons.

1.1.3 Difficulty of Parallel programming

With the increasing demands of real-time large volumetric data processing, parallel computing, especially based on multi-core architectures, can be employed to further improve total performance of real-time voxelization.

As multi-core architectures gain widespread use, it becomes increasingly important to be able to harness their additional processing to achieve higher performance. However, exploiting parallel cores to improve single-program performance is difficult from a programmer's perspective because most existing programming languages dictate a sequential method of execution. Parallel programming on multi-core processors has become the industry's biggest software challenge.

Likewise, as multi-core graphics processors have emerged as a commodity platform for parallel computing, graphics processors deliver their dramatic high performance through a well-balanced, carefully managed, highly parallel architecture. However, the developer requires the knowledge of the Graphics Processor Unit (GPU) architecture and much effort in developing GPU applications. Such difficulty is even more for complex and performance-centric tasks. Current parallel development on Compute Unified Device Architecture (CUDA) [CUDA08a] for mutli-core GPUs has shown such extra time and effort cost. Even so has the latest Open Computing Language (OpenCL) [OCL08] for general or heterogeneous multi-core systems.

1.2 Motivation

There are two main motivations as follows:

1) More and more volume graphics applications need fast and multi-valued voxelization technologies.

All of the current applications and potential research involving volumetric representations provide a strong motivation to propose an approach for interior data acquisition of 3D geometric objects (i.e., solid voxelization) in real time. Furthermore, Multi-valued voxelization allows direct volume rendering of voxelized geometry as well as volumes with intermixed geometry, accurate multi-resolution representations, and efficient anti-aliasing. Prior voxelization methods either computed only a binary voxelization or inefficiently computed a multi-valued voxelization. Although multi-valued solid voxelization is in general an expensive process, hardware accelerated techniques significantly improve the efficiency of the voxelization process. As real-time volume rendering hardware becomes widely accessible, the research emphasis should be on interactive voxelization, which clearly bridges volume graphics and practical applications.

2) Very few parallel voxelization methods have been studied; development on current multi-core GPU architectures becomes the biggest software challenge.

With the increasing demands of real-time large volumetric data processing, many parallel and large-scale data volumetric rendering approaches have been studied intensively [Sto03][DeM03]. However, very few parallel voxelization algorithms have been investigated so far.

Multi-core architectures have started a new era of computing and boost performance and efficiency of parallel programs, however parallel programming on multi-core processors has become the industry's biggest software challenge. Moreover, most of the existing application base is still composed of sequential applications. How to execute those sequential programs efficiently and reliably on multi-core systems is still remaining

a critical and challenging problem. If we can change the hardware to eliminate the constraints, most of the existing single-processor software programs may be able to be directly used with nearly no change. In this dissertation research, a new parallel computing approach using multi-core pipelining is proposed to further improve total performance of real-time solid voxelization. This multi-core architecture can be extended to more general multi-core GPU, multi-core CPU, and multi-core/many-core heterogeneous systems.

1.3 Objective

The research in this dissertation is to develop the approaches to reach the following goals:

- Real-time

  to fast voxelize a solid geometric object into a multi-valued volumetric object in real time. Not only can using multi-core parallel methods reach this goal, but also a new approach based on a single-CPU/GPU (i.e. one-core) system does so.

- Diverse

  to dynamically generate a volumetric object to acquire as many diverse internal features as possible during the voxelization.

- Error-free

  to reduce as many limitations or errors as possible, such as low speed, missing voxels, prolonged lines, binary-only voxels, and non-transparency.

- Efficient and Extensible

  to be efficient, robust, flexible, and to be convenient to integrate into a wide range of other applications.

- Easy to implement

  to be easy to implement on both a single-GPU/CPU system (i.e. one-core system) and multi-core parallel system, which need a novel parallel architecture without changing one-core software to be designed to further improve total performance

In summary, the technical objective of this work is to apply the proposed methods to a wide range of volume graphics applications and to be extended to more general multi-core parallel systems.

1.4 Proposed Solutions

To reach all the above goals, this dissertation proposes a core method for GPU-accelerated real-time solid voxelization and a novel generic parallel architecture using multi-core pipelining based on program slicing by forced interrupts. It also proposes a multi-core pipelined parallel solid voxelization system in combination with this method and the new architecture. Additionally, it uses this new real-time solid voxelization method to present several fast interactive approaches for two important volume graphics applications: volume modeling and collision detection.

From a functional point of view, the proposed research solution is composed of three interrelated components as follows:

- First Component

  The first component is a real-time solid voxelization by GPU acceleration based on a conventional single-CPU computer with a 3D accelerator (GPU). The key dynamic solid slice function mechanism and masking techniques are used for creating the various kinds of solid slices in a volume. This method can significantly improve the speed of multi-valued solid voxelization. Theoretically, this approach can

generate various kinds of volumetric models based on inputs to different slice functions. By dynamically applying different slice functions, any surface-closed geometric model can be voxelized into a solid volumetric representation with diverse multi-valued interior features, such as rainbow, marble, wood, and translucent jade. The algorithms are easy to implement and convenient to integrate into diverse applications, such as volume rendering, volume modeling, collision detection, medical simulation, 3D printing, and computer art.

- Second Component

The second component is an effective scheme for real-time parallel solid voxelization using multi-core pipelining. It comprises of two parts: 1) one is a novel generic parallel architecture using multi-core pipelining based on program slicing by forced interrupts and a distinct memory/cache dynamic management technology; and 2) the other is a multi-core pipelined parallel solid voxelization system by extending this novel architecture to a special multi-core pipelined GPU architecture. The new architectures provide simple and effective solutions to on-the-fly computations by transferring the operating states from processor to processor. The most important advantage is that they require practically the same software as is currently used on a single-CPU system, instead of conventional parallel computing methods, such as threading, load balancing, and scheduling. Through this pipeline architecture, the solid voxelization program for a conventional single-CPU computer with 3D accelerator (GPU) is automatically scheduled onto multiple processors with interrupt control. Consequently, the speed, quality, and size of the voxelization can be improved significantly.

- Third Component

The third component is the case studies for two important applications in volume graphics, volumetric CSG modeling and volumetric collision detection, using real-time GPU-accelerated solid voxelization. First, a new algorithm of fast volumetric CSG modeling is proposed for volume scene by scene computation. It is capable of carrying out the interactive modeling of heterogeneous volume scene. Second, a simple and fast collision detection method is presented as one of the successful applications using this core voxelization approach. It can improve the collision detection performance as well as detection accuracy based on the voxel level. Both corresponding experimental results will demonstrate the effectiveness, flexibility, and diversity of the developed approaches.

1.5 Contributions

The main contributions of this dissertation are:

- A new real-time solid voxelization approach by GPU acceleration

  A real-time solid voxelization approach is developed using dynamic solid slice functions and masking techniques based on GPU acceleration to convert solid geometric objects into any size of diverse multi-valued volumetric models at the same time. This approach significantly improves not only the voxelization speed for solid geometric objects in real time but also the quality with error-free volumetric models. In addition, the algorithm is slice-independent voxelization processing, which is critical for any parallelization approaches.

- A novel multi-core pipelined parallel architecture by forced interrupt

A novel generic parallel architecture using multi-core pipelining is developed based on program slicing by forced interrupts and a new distinct memory/cache dynamic management technology for parallel computing. Through this architecture, most of the existing single-processor software programs can be directly used with nearly no change and reduce the extra parallel programming cost of time and effort at maximum. This new parallel architecture can also be successfully applied to other multi-core/many-core architectures or heterogeneous systems.

- A new parallel solid voxelization system using multi-core pipelining

  An effective on-the-fly parallel scheme for real-time solid voxelization is developed based on the new GPU-accelerated real-time solid voxelization approach and the extension of the novel multi-core pipelined architecture to multi-core GPUs. The parallel computing in this system can be reached only using the same solid voxelization software as those on the sequential computer with one GPU. This new multi-core pipelined parallel system further improves total performance of real-time solid voxelization. It can also be extended for other applications without changing the sequential software for parallel graphics and visualization.

- Helping to make interactive voxelization-based applications possible and expanding more various applications

  - **Volume Modeling**: A new application algorithm for fast volumetric CSG modeling by volume scene tree computation developed based on the core real-time solid voxelization approach. This algorithm is

capable of carrying out interactive modeling of heterogeneous volume scene on the GPU.

- **Volumetric Collision Detection**: A new application algorithm for simple and fast volumetric collision detection developed based on the core real-time solid voxelization approach.

1.6 Document Organization

Chapter 2 reviews past research in various domains including volume graphics, solid voxelization, Constructive Solid Geometry (CSG), computer graphics hardware and APIs, parallel computing, multi-core GPU architectures, multi-core programming, and other relevant fields. Chapter 3 describes GPU-based real-time solid voxelization including the definitions of solid objects and solid slice function, the principles and algorithms of dynamic solid slice creation and real-time solid voxelization by GPU acceleration. Some experimental results, performance analysis, and comparison are given in detail. Chapter 4 presents a novel generic parallel architecture using multi-core pipelining based on program slicing by forced interrupts and a distinct memory/cache dynamic management technology. Chapter 5 describes a new parallel solid voxelization system using the new multi-core pipelined architecture with its extension to multi-core GPUs. Both detailed theoretical and experimental performance analysis are given as well. Chapter 6 and Chapter 7 are two important case studies using developed real-time solid voxelization approaches. In Chapter 6, Case Study 1, volumetric CSG modeling by real-time solid voxelization is discussed. This chapter includes detailed volumetric scene tree design and tree pruning optimization. In Chapter 7, Case Study 2, volumetric collision detection by real-time solid voxelization is discussed, including collision detection color coding and

fast collision check. Both case studies provide the experimental results. Chapter 8 concludes this dissertation and discusses the future research directions.

1.7 Overview of the Proposed Systems

The overview of the proposed systems is shown in Figure 1.3.

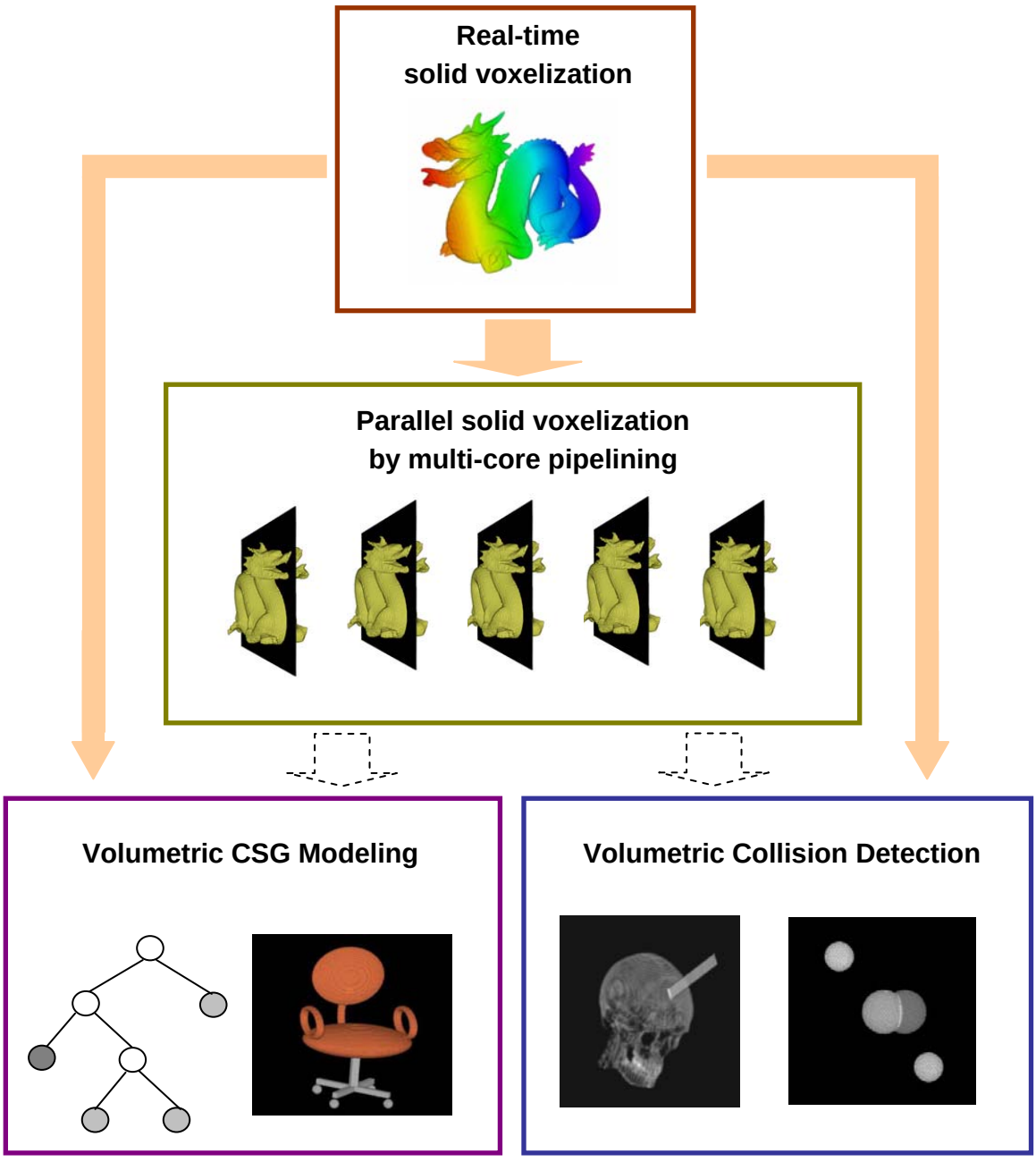


Figure 1.3: Overview of proposed systems

2 Background

The background material starts with a brief overview of concepts in volumetric data, volume graphics. The modern graphics hardware and APIs are also introduced including GPU, graphics hardware pipeline and visualization. Then, the prior work on solid voxelization is described. Recent existing solid voxelization algorithms are classified into six categories to compare. The noise functions and volumetric modeling are introduced as well. Finally, parallel computing, multi-processor pipeline, multi-core architectures, multi-core programming are described.

2.1 Volume Data

Volume data are first defined formally by Kaufman in 1980's. It was defined by "3D entities that may have interior information, may not consist of surfaces and edges, or may be too voluminous to be represented geometrically" in [Kau93] and [Kau96]. In other word, volume data can be represented by a mass of discrete points in regular or irregular form. They can be obtained by sampling, simulation, or modeling techniques.

More specifically, Kaufman gave detailed descriptions for volume data as follows:

"Volume data is typically a set *S* of samples *(x, y, z, v),* representing the value *v* of some property of the data, at a 3D location *(x, y, z).* If the value is simply a 0 or a 1, with a value of 0 indicating background and a value of 1 indicating the object, then the data is referred to as binary data. The data may instead be multi-valued, with the value *v*

representing some measurable property of the data, including for example, color, density, heat or pressure. The value *v* may even be a vector, representing for example, velocity at each location."

In general, a regular grid in 3D space, 3D array, is typically used to store the values of the samples. The location of each element indicates the position of the sample on the grid. This kind of the grid is also called *volume buffer* or *cubic frame* buffer in volume graphics.

2.2 Volume Graphics

Volume graphics processes the graphics scenes defined by volumetric data [Kau93]. It includes the synthesis, modeling, manipulation, and rendering of volumetric objects, stored in a volume buffer. More importantly, volume-based graphics offers a consistent solution to the primary deficiencies of the traditional surface-based graphics, including its inability to encapsulate the internal description of a model, and the difficulties in representing and rendering amorphous phenomena (e.g. cloud, gas, fluid, etc.).

Volume graphics also proposes to extend raster graphics into 3D by extending a 2D frame buffer into a 3D volume buffer. The fundamental technique, upon which a volume graphics system is built, is the real-time rendering volume data from the volume buffer to a 2D frame buffer displayed on the screen. The breakthrough was made in the last few years with the release of the VolumePro real-time volume rendering chip by Mitsubishi [Pfi99]. Real-time volume rendering and other related software and hardware processes have made volume graphics system a feasible alternative to traditional graphics engines, and provided the necessary computational and rendering power for a volumetric paradigm in 3D interaction.

2.3 Computer Graphics Hardware and APIs

2.3.1 3D Graphics APIs

Nowadays, all 3D programming applications are developed based on two major 3D programming interfaces, OpenGL [OGLlink, Woo99] and Direct3D [D3Dlink], to communicate the rendering commands to GPUs instead of implementing 3D graphics rendering algorithms in software using CPUs. OpenGL, developed by Silicon Graphics [SGI], is an open standard for 3D programming for Windows, Linux, UNIX, and Macintosh computers. Direct3D is an evolving set of Microsoft multimedia programming interfaces for 3D programming. Direct3D is an evolving set of Microsoft multimedia programming interfaces for 3D programming.

Over the last decade, OpenGL has evolved along with graphics hardware. OpenGL is extensible, meaning that OpenGL implementers can add new functionality to OpenGL in an incremental way. Today, scores of OpenGL extensions provide access to all the latest GPU features. This includes ARB-standardized extensions for vertex and fragment programmability. The current version of OpenGL work includes both assembly-level and high-level programmable interfaces.

2.3.2 GPU

A graphics processing unit or GPU is a dedicated graphics processing and rendering device for a computer. A GPU implements a number of graphics primitive operations in a way that makes running them much faster than drawing directly to the screen with the host CPU. Modern GPUs are very efficient at manipulating and displaying computer

graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for a range of complex algorithms.

In the old days of 3D graphics on a PC (before there were GPUs), the CPU handled all the vertex transformation and pixel-pushing tasks required to render a 3D scene. The graphics hardware provided only the buffer of pixels that the hardware displayed to the screen. These graphics systems were specifically designed by the companies such as Silicon Graphics (SGI) and Evans & Sutherland (E&S). These systems were very important to the historical development of computer graphics, but because they were so expensive, they did not achieve the mass-market success of single-chip GPUs designed for PCs and video game consoles. Additionally, they only support OpenGL.

So far, five generations of GPU evolution [Fern03] can be identified in the industries. Each generation delivers better performance and evolving programmability of the GPU feature set. The five generations of GPU evolutions are summarized as follows:

- First GPU Generation -- Rasterization

  The first GPU on PC should be traced back to early 1990's. 3DLabs was an early pioneer in bringing 3D graphics to the PC. Its GLINT 300SX graphics processor was the industry's first single chip in 1994. The first generation of GPUs (1994-1998) includes 3D Labs' GLINT SX/TX series, E&S's Galaxy, NVIDIA's TNT2, ATI's Rage, and 3dfx's Voodo03. These GPUs are capable of rasterizing pre-transformed triangles and applying textures. However, these GPUs lack the ability to transform vertices of 3D objects and the adequate set of math operations to compute the pixels for combining textures.

- Second GPU Generation -- Vertex transformation and Lighting

The second generation of GPUs (1999-2000) includes 3D Lab's WildCat, NVIDlA's GeForce 256 and GeForce2, ATI's Radeon 7500, and 53's 5avage3D. These GPUs offload 3D vertex transformation and lighting from the CPU. Fast vertex transformation was one of the key capabilities that differentiated high-end workstations from PCs prior to this generation. The set of math operations for combining textures and coloring pixels is still limited. That is, it is still not truly programmable.

- Third GPU Generation -- Vertex Programmability

The third generation of GPUs (2001) includes NVIDlA's GeForce3 and GeForce4 Ti, 3D Lab's WildCat and Oxygen, Microsoft's Xbox, and ATI's Radeon 8500. This generation provides vertex programmability rather than merely offering more configurability. These GPUs support vertex programmability but lack true pixel programmability.

- Fourth GPU Generation -- Vertex and Pixel Programmability

The fourth generation of GPUs (2002 and 2005) includes NVIDlA's GeForce FX family with the CineFX architecture and ATI's Radeon 9700. These GPUs provide both vertex-level and pixel-level programmability. This level of programmability opens up the possibility of offloading complex vertex transformation and pixel-shading operations from the CPU to the GPU. This is the generation of GPUs where most shading languages, such as Cg, get really interesting.

- Fifth GPU Generation -- GPGPU and multi-core GPU

The fifth and current generation of GPUs (2005 and on) provides a new concept to use a modified form of a stream processor to allow a general purpose graphics processing unit (GPGPU). It turns the massive floating-point computational power of a modern graphics accelerator's shader pipeline into general-purpose computing power. This kind of GPU integrates many streaming processing elements (i.e. cores) for parallel processing. More multi-core GPU architecture background will be presented in Section 2.10.

2.3.3 Graphics Hardware Pipeline

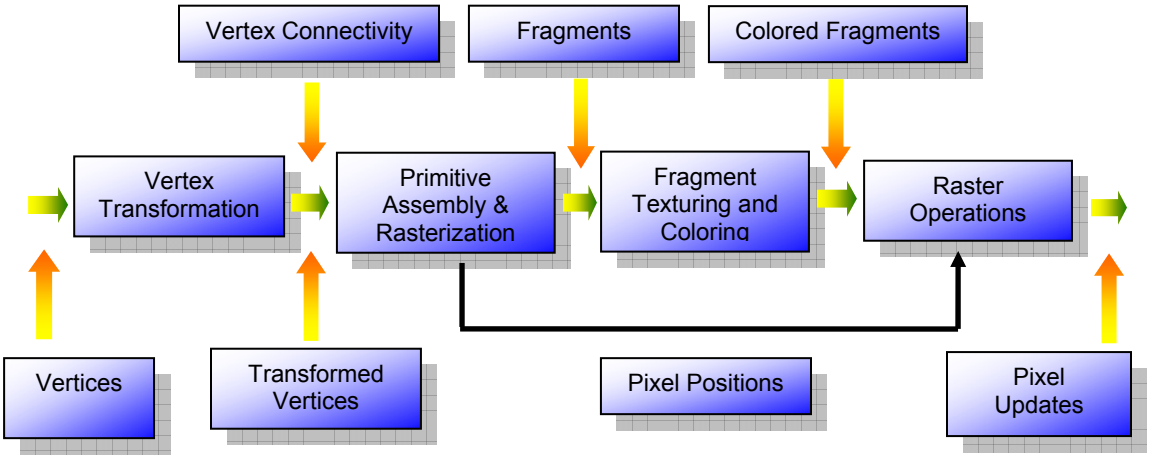


Figure 2.1: The graphics hardware pipeline

A conventional graphics hardware pipeline processes a multitude of vertices, geometric primitives, and fragments in a pipelined fashion. Figure 2.1 shows the graphics hardware pipeline used by today's GPUs [CemC03]. The 3D application sends the GPU a sequence of vertices batched into geometric primitives: typically polygons, lines, and points.

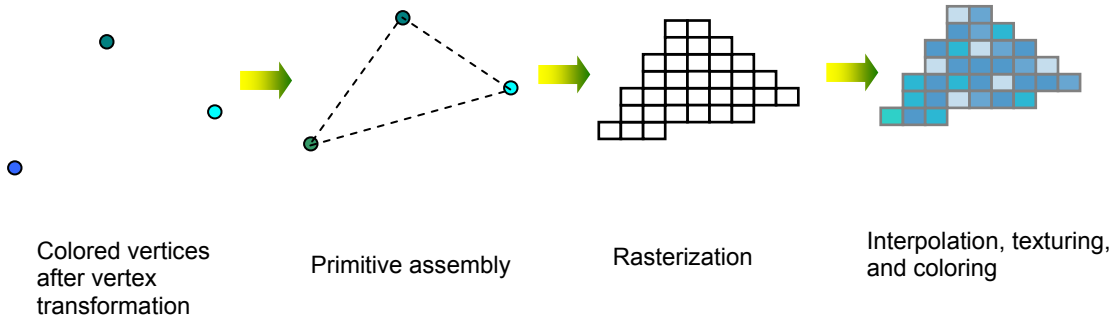


Figure 2.2: The graphics pipeline

Figure 2.2 depicts the stages of the graphics pipeline [CemC03]. That is to say, it shows how triangles are rasterized in the graphics pipeline. As the figure indicates, there are four stages to raster triangles as follows:

1) transforms and colors the vertices
2) assembles primitives to create the triangles from the vertices
3) rasterizes (i.e. "fill in") the triangles with fragments
4) applies interpolation, texturing, and coloring.

In this dissertation, the rasterization by graphics hardware is very important for the new proposed real-time solid voxelization approach.

2.4 Solid Voxelization

Voxelization is the process of generating volume dataset from a synthetic geometric model, and is a necessary step in the volume graphics pipeline as proposed in [Kau93]. Conceptually, voxelization is a set membership classification problem for all voxels in a volume against the given 3D model. In addition, from the view of the process, voxelization is a 3D scan conversion [Kau91a] process.

In early years, a number of curve and surface voxelization algorithms have been proposed [Kau86, Kau87a, Kau87b, Kau88, Coh91, Coh95, Coh97, Hua98, Dan70, Mok88, Dan92, Hua98]. Besides this large amount of such research, the extensive surface voxelization studies have still been continued in recent years [Sra99, Dac00, FC00, Don04, Hsi05, Eise06]. Broadly speaking, these algorithms aim to provide extensions of 2D scan conversion methods to a volumetric domain, but are generally not fast enough for interaction. The voxelization of solid models is, however, a much harder problem. This is because solid objects are normally represented by their boundary surfaces without explicit interior information, which would require an inside test for each voxel in the volume space.

2.4.1 Prior Work of Solid Voxelization

Although less research has been done on solid voxelization than surface voxelization in volumetric modeling and representation in a volumetric domain, the prior work related to solid voxelization has been certainly studied over the past years, but they are still limited.

In early years, a general purpose algorithm as mentioned in [Lee82] based on point classification is too slow. Traditional 3D scan-conversion approaches for voxelization were studied thoroughly [kau86, Kau88]. Later, some algorithms based on filtering and distance volume mention to use different filtering or distance values to fill all interior voxels or to thicken the boundaries of the polygons to avoid aliasing [Wan93, Bre98, Sra98, Sra99, Sra00, Pas03, Hsi05].

Additionally, a new, morphological criterion was presented in [Bae00] for determining whether a geometric solid is suitable for voxelization at a given resolution.

The criterion embodies two conditions, namely that the curvature of the solid must be bounded and the critical points of the distance field must be at a certain distance from the boundary of the solid. For solids that fulfill this criterion, they presented an analytic and an empirical bound for the trilinear reconstruction error.

In recent years, more and more studies try to explore the benefits of graphics hardware for more efficient rendering. Some fast hardware-based techniques have been employed for solid voxelization.

A slice-dependent algorithm for solid voxelization is presented by Chen and Fang [Che98, CF00] by means of setting one slice-thick projection as a clipping plane to generate a voxel-based closed surface boundary and then applying a logical XOR operation between the current slice and the previous slice that has already been classified against the solid object. However, only a binary volume is generated and some voxels may be missing due to XOR operations. The problem of missing voxels leading to prolonged lines also exists. Some such experimental image results are shown in Figure 3.12. Another limitation of this algorithm is slice-dependent voxelization, which does not suit for most parallelization methods.

Later on, a new slice-independent solid voxelization algorithm is proposed based on stencil masking techniques in [Lia01]. Stencil buffer is a core technique used for the important masking techniques in the solid voxelization algorithm. Although stencil buffer technique has been used widely in many computer graphics applications [OGL07], such as shadow volumes, surface-based Constructive Solid Geometry (CSG) rendering, it was first used for solid voxelization in [Lia01] and [Lia02]. In this algorithm, a stencil buffer is utilized as a mask to generate interior voxels while solid geometric objects are clipped

by a clipping plane without setting a thick projection for each slice. However, this algorithm needs to draw the object twice to count the number of front faces and back faces, respectively. Additionally, only one single color volume is generated. The total generation speed is influenced by 2D texture mapping due to the large amount of memory swapping between GPU and CPU. This dissertation improves and extends this technique to combine stencil buffer with GPU techniques to first generate the dynamic and diverse solid volumes [Lia08a].

Karabassi and Theoharis [Kar99] presented the use of six z-buffers to project an object to six faces of its bounding box for the outermost parts and read back the information from depth buffer to synthesize the volume. However, it works well on convex objects only. Also, it take much time to generate the inside voxels due to using pure software methods to check each voxel. Passalis et al. [Pas04] improved Karabassi et al.'s algorithm to handle non-convex objects and use double layer buffering techniques to determine the number of actual intersections per voxel. However, this algorithm becomes slow as the number of z-buffers increases.

Varadhan et al. [Var03] used the max-norm distance computation algorithm to design a reliable voxel-intersection test to determine whether the surface of a primitive intersects a voxel for solid voxelization and generate adaptive distance fields with the guarantee of Hausdorff distance. However, the overall algorithm runs slowly even with rasterization hardware for local refinement. Hsieh et al. [Hsi05] presented a slice-based approach for distance field generation and voxelization by GPU and tested for surface voxelization. However, due to the bottleneck of pixel shader performance, the fast solid voxelization using this algorithm has not been fully tested and guaranteed.

Dong et al. [Don04] proposed a real-time voxelization method using GPU acceleration to rasterize and texelize an object into three directional textures and then synthesize these textures back to the final volume. This algorithm is designed for surface voxelization but can be extended to solid voxelization, which uses a 3D scan-filling operation along three axis directions and then check for common voxels. However, prolonged lines appear where some voxels are missing at the rasterization stage. The 3D scan-filling operation is time-consuming. Eisemann and Decoret [Eis06] employed similar GPU-based voxel encoding techniques for real-time voxelization. However, only a subset of voxels, instead of the entire interior voxels, is generated without any flood-fill performing to classify interior and exterior voxels. In addition, some voxels are missing as aligned with the view direction and some are not voxelized continuously due to a large slope in z direction. Strictly speaking, this voxelization algorithm is not pure solid voxelization.

2.4.2 Classification and Comparison of Existing Solid Voxelization Algorithms

The above solid voxelization algorithms can be classified into the following categories:

Category 1: Pure software-based solid voxelization

This kind of algorithm uses pure software methods to generate each voxel in a volume domain. Such typical algorithms include Kauffman et al.'s traditional 3D scan conversion [Kau86, Kau88] and Baerentzen et al.'s morphological approach [Bae00].

Category 2: Voxel encoding and filling based solid voxelization

This kind of algorithm first voxelizes a closed surface, and then fills the interior voxels using 3D scan-filling or floodfill with GPU acceleration. This category of algorithms includes Dong et al. [Don04] and Eisemann et al. [Eise06].

Category 3: Multiple z-buffer based solid voxelization

This kind of algorithm first projects an object onto each z-buffer for multiple views, and then checks each interior voxel with or without GPU acceleration. This category of algorithms includes Karabassi et al. [Kar99, Pas04].

Category 4: Slice-dependent solid voxelization

This kind of algorithm first voxelizes a closed surface boundary in each slice, and then fills the interior voxels of each slice by a XOR operation or 2D scan-filling. Chen and Fang's algorithm [Che98, CF00] is a typical one.

Category 5: Slice-independent solid voxelization

This kind of algorithm directly voxelizes solid geometric objects on each slice. My stencil masking algorithm is a typical one. More details will be discussed in this dissertation.

Category 6: Distance-field based solid voxelization

This kind of algorithm uses a distance field for solid voxelization. Recent Varadhan's [Var03] and Hsieh et al.'s methods [Hsi05] are such algorithms.

In a summary, several typical solid voxelization algorithms are comprised briefly as shown in Table 2.4.1.

Table 2.4.1: Comparison of several existing typical solid voxelization algorithms

Category	Speed	Anti-aliasing	Accuracy	Hardware Acceleration	Voxel Color Channel	Other Limitations
1	extremely slow	can use algorithms to generate	depends on algorithms	No	From binary to RGBA (1-bit to 32-bit)	Due to pure software computation, the running time is extremely slow.
2	Slow	No	Low	Yes	Binary (1-bit)	1) Need to fill interior pixels by GPU or non-GPU 3D scan-filling or flood filling methods. This leads to slow processing. 2) Prolonged lines appear when some voxels are missing at the rasterization stage.
3	slow	No	Low	Software and hardware mixing	Binary (1-bit)	voxel missing, erroneous voxels, incontinuous voxelization
4	Related to the number of slices	No	Low	Yes	Binary (1-bit)	Prolonged lines appear when some voxels are missing at the rasterization stage. Time complexity is related to number of slices.
5	Related to the number of slices	No	Higher if size of volume is bigger	Yes	From binary to RGBA (1-bit to 32-bit)	Time complexity is related to number of slices.
6	Slow	Yes	Yes	Software and hardware mixing	Luminance (8-bit)	Voxelization speed is slow and voxel channel is only 8-bit.

2.4.3 Parallel Voxelization

With the increasing demands of real-time large volumetric data processing, many parallel and large-scale data volumetric rendering approaches have been studied intensively [Sto03][DeM03]. However, very few parallel voxelization algorithms have been investigated so far. Stolte and Kaufman ever proposed an efficient parallel recursive voxelization method based on spatial recursive subdivisions and data structures on SGI Challenge shared-memory multiprocessor system [Sto98]. This is still a conventional parallel programming method, which need to consider data distribution, load balancing, etc.

In this dissertation, a total different parallel voxelization system will be proposed based on multi-core pipelining without changing the sequential programs. In the other word, this new novel system does not need complicated parallel programming unlike the existing multi-core systems.

2.5 Noise Functions

Noise functions used in computer graphics are smoothly varying functions taking $R^n \rightarrow [-1, 1]$, for at least $n = 1, 2, 3$, without obvious repetition [Pha04]. One of the most crucial properties of a practical noise function is that it is band-limited with a known maximum frequency. This makes it possible to control the frequency content added to a texture due to the noise function so that frequencies higher than those allowed by the Nyquist limit are not introduced.

Many noise functions are built on the idea of an integer lattice over R^3. First, a value is associated with each integer position in space. Then given an arbitrary position in

space, the eight adjoining lattice values are found. These lattice values are then interpolated to compute the noise value at the particular point.

Perlin noise is a famous noise function introduced by Ken Perlin [Per85, Per02]. It has a value of zero at all integer lattice points. Its variation comes from gradient vectors at each lattice point that guide the interpolation of a smooth function in between the points. This noise function is computationally efficient and easy to implement.

David Ebert [Ebe03] uses various noise functions to develop newer texturing techniques such as cellular texturing, which can be used, for example, to create convincing stone patterns. He also studies on animating solid textures (e.g. marble forming, volumetric gasses, etc.), fractal terrain generation, and utilizing existing graphics APIs and hardware for real-time procedural texturing.

2.6 Constructive Solid Geometry

A technique often used for 3D model composition is the Constructive Solid Geometry (CSG) representation. It allows users to define complex 3D solid objects by hierarchically combining simple geometric primitives using Boolean operations and affine transformations [Req80, Hof89]. It is a very popular and powerful solid modeling scheme, and is particularly suitable for interactive object manipulations and design. Traditionally, CSG primitives are defined by simple analytic objects, such as cubes, cylinders and spheres. Some recent CSG algorithms can also support primitives that are generic solid models defined by their boundary surfaces.

The CSG technique was originally introduced in solid modeling for computer aid design applications [Hof89], and has also been widely used later in 3D computer graphics and animation [Gol89] [Wie96][Rap97]. A natural extension of the CSG method is to

allow volume datasets to be included as primitives in the CSG construction process, as described independently in several publications [Wan94][Fan98a][Che00] [FC00].

2.7 Volumetric CSG Model

The ability to integrate and process volumetric information has recently become increasingly desirable in mainstream computer graphics. In volume graphics, volume modeling and volume rendering are two major technologies. Volume rendering has been deeply studied and achieved great developments since the volume concept was proposed several decades ago. Recent advances in real-time volume rendering hardware systems, such as the VolumePro chip by Mitsubishi [Pfi99], have made volume techniques a promising direction in 3D graphics and 3D modeling. Volume modeling, however, is still in its infancy with few truly effective volume modeling techniques and limited research activities in the 3D modeling community [Gre00].

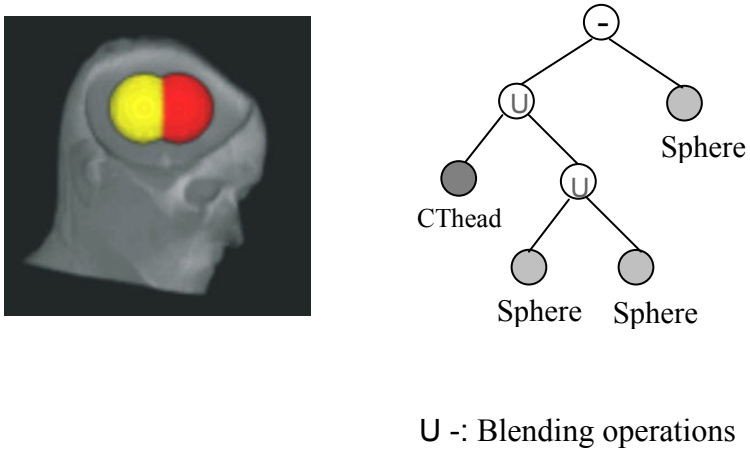


Figure 2.3: An example of VCSG model

Volumetric CSG (VCSG) modeling is one of the important volume modeling methods. Using voxel-based volume representations, a further extension can include objects extracted from volume data sets using intensity thresholding. These volume datasets may come from various types of the scanning of real objects, such as CT, MRI, and microscopy images, or from the sampling of implicit or procedural functions. Such extended CSG models are sometimes called *volumetric CSG models* [Fan98a][Fan00], and are very useful in applications such as medical imaging, surgical simulation, and amorphous phenomenon modeling. Figure 2.3 shows an example of the VCSG model.

Due to the lack of explicit representation of surface boundaries, displaying CSG is not directly supported by standard graphics systems. Although several interactive CSG rendering algorithms have previously been developed [Gol89][Rap97][Thi87][Wie96], they cannot be directly applied when volume data sets are involved. In principle, there are two potential solutions for the rendering of VCSG models. The first is to explicitly extract an iso-surface from the volume data set, and convert the model into a regular geometric CSG model, which can then be displayed by an existing CSG rendering algorithm. This method is, however, very slow and difficult for two reasons: (a) the iso-surface extraction process is time-consuming, and (b) the extracted iso-surface normally contains an extremely large number of tiny polygons, which are difficult and expensive to display and operate. The second solution is to first convert the CSG model into a voxel-based volume representation, and then display the resulting volume using a standard volume rendering algorithm. This solution, however, requires a fast volume conversion (voxelization) process for VCSG models.

2.8 Parallel Computing

In general, there are three major approaches used for multiprocessor processing:

1) Data-parallel: partitions data and schedule them onto the multiple processors,

2) Task-parallel: Partitions a program into functions/tasks and schedule them onto the multiple processors,

3) Pipeline-parallel: decomposes a program and run each state simultaneously on sequential elements of the data flow.

The first approach is suitable for the data-independent circumstance. However, the scheduling may be complicated depending on the application programs.

For the second method, in practice, it is often difficult to divide a program in such a way that separate CPUs can execute different portions without interfering with each other. Furthermore, this type of parallel processing requires very sophisticated software called distributed processing software [Cul98][Gra03][Jor03].

For the third parallel computing method, a pipeline is common paradigm for very high-speed computation. A pipeline provides high speed because it supports the parallel execution of many operations. The pipeline parallelism allows for parallelization of a single task when there is a partial or total order in the dataset implying the need for state and therefore preventing the use of data parallelism. This approach is limited by the sequential decomposability of the task and the length of the longest stage.

2.9 Multi-processor Pipeline by Forced Interrupts

Another promising parallel computing method is based on the multi-processor pipeline architecture by dividing the program in equal duration by forced interrupts as

described in [Ber90][Ber94][Ber00]. This technology has US PATENT No. 6145071 issued in 2000 and owned by The George Washington University. It can automatically schedule the program onto the multiple processors. The architecture processes an information flow progressively in a helicoidal pattern by relocating portions of incoming data. This pattern ensures that the incoming data flow will not be interrupted.

The multi-processor pipeline allows an arbitrary algorithm to be performed on-the-fly on a data chunk, given a sufficient number of processors. If an algorithm can be performed by a conventional microprocessor under static conditions, it can be performed on the multiprocessor pipeline. Another advantage of this architecture is to use practically the same software as a sequential computer and to be able to continuously process the intensive information flows.

This multi-processor pipeline is a simple and effective solution to the problem of uninterrupted processing of intensive flows of information. This technique has been proposed to be used for effectively processing the challenging problem of very intensive continuous flows of data [Ber92].

However, there are several limitations for some applications. Frequent data relocation especially for large data block applications like 3D graphics can cause big overhead costs leading to the overall performance decrease. The system described in [Ber94] uses data overlapping to solve the problem of processed data chunks across the segmentation imposed by the buffer size. The amounts of memories of the processors have to be occupied by these duplicate (i.e. overlapped) data. Moreover, to reduce the bus traffic, the data relocations, i.e. the loadings/unloadings, are arranged so that only one data chunk relocates in one shared bus at a particular time. Additionally, for the data required longer

pipeline to process, the pipeline need Special handling, such as overflow facility or accumulation and then sending back while the data stream ceases.

In this dissertation research, the core generic forced-interrupt driven multiprocessor pipelining technique is still employed. But, a novel multiprocessor architecture is proposed with a new memory/cache management technology to significantly overcome all the above limitations of the original multiprocessor pipeline.

2.10 Multi-core GPU Architectures

Since GPUs were originally designed with special purpose for 3D graphics applications. The hardware GPU architecture differs from multi-core CPUs significantly. Current GPUs consists of many SIMD (Single Instruction Multiple Data) multi-processors whereas current multi-core CPUs offer a much smaller number of cores. Moreover, most GPUs do not support atomic operations or locks. The GPU supports thousands of concurrent threads. GPU threads have low context-switch and low creation time as compared to CPU threads. The threads within a thread group share the computation resources (such as registers) on a multiprocessor. A thread group is divided into multiple schedule units that are dynamically scheduled on the multiprocessor. The device memory of current GPUs is of both high bandwidth and high access latency.

The latest Intel's multi-core graphics chip, also known as Larrabee [Sei08], is one of the boldest graphics projects in the world and offers full compatibility with DirectX and OpenGL APIs. Moreover, the chip will also be capable to process the entire x86 instruction set that comes implemented into modern processor architectures.

However, all of the current multi-core GPU architectures need threading-based parallel programming. In this dissertation, I will use a completely different way to design a multi-core GPU architecture without parallel programming.

2.11 Crossbar-based Technologies

The crossbar switch [Cro08] has been used in many areas like telephone exchange. As computer technologies have improved, crossbar switches have found uses in systems such as the multistage interconnection networks that connect the various processing units in a Uniform Memory Access (UMA) parallel processor to the array of memory elements. Crossbar switches have been also designed to line entire computer systems as well.

Crossbar-based memory architecture has been used on mainframe computers to increase memory bandwidth in multi-processor systems since decades. The companies such as Unisys, SGI, and Sun, have brought the technology down to the server and workstation platforms.

Nvidia patented Lightspeed Memory Architecture^{TM} (LMA) [LMA01] has employed the use of a crossbar to maximize the efficiency of data transfer between the graphics processing unit and the graphics memory on the GPU. A memory crossbar can eliminate bottlenecks associated with existing memory architecture as it replaces the conventional system bus architecture. Instead of sharing a bus, communication between the processor and the memory uses dedicated connections.

In this dissertation, a distinct dynamic memory/cache management technology using crossbar techniques is proposed for the new multi-core architectures.

2.12 Multi-core GPU Programming

CUDA (Compute Unified Device Architecture) [CUDA08a, CUDA08b] is a software platform for massively parallel high-performance computing on the powerful GPUs. It is architected to support all computational interfaces, including C and new open standards like OpenGL [OGL05] and DirexX11 [DTX08] Compute. Although CUDA automates thread management, it does not relieve developers entirely from thinking about threads. Developers must analyze their problem to determine how best to divide the data into smaller chunks for distribution among the thread processors. This data layout or "decomposition" does require programmers to exert some manual effort and write some explicit code. CUDA isn't the only option for software development on Nvidia GPUs. CUDA is merely Nvidia's platform.

RapidMind's Multicore Development Platform supports multiple processor architectures, including Nvidia's GPUs, ATI's GPUs, IBM's Cell BE, and Intel's and AMD's x86. This flexibility lets developers target the architecture currently delivering the best performance—without rewriting or even recompiling the source code. RapidMind [Rap08]'s C++ extensions appear less dependent on the processor's micro-architecture than CUDA's extensions are.

OpenCL (Open Computing Language) is an open industry standard for general-purpose parallel programming of heterogeneous systems. OpenCL was conceived by Apple Inc., and established as standard by Khronos Group in cooperation with others. The purpose is to recall OpenGL and OpenAL, which are open industry standards for 3D graphics and computer audio respectively, to extend the power of the GPU beyond graphics (GPGPU). OpenCL provides a uniform programming environment for software

developers to write efficient, portable code for high-performance compute servers, desktop computer systems and handheld devices using a diverse mix of multi-core CPUs, GPUs, Cell-type architectures and other parallel processors such as DSPs.

3 Real-Time Solid Voxelization by GPU Acceleration

This chapter presents a GPU-accelerated slice-independent solid voxelization approach that utilizes a dynamic slice function mechanism and masking techniques to significantly improve solid voxelization speed in real time as well as create various multi-valued and high-quality solid volumetric models with different slice functions. In particular, by dynamically applying different slice functions, any surface-closed geometric model can be voxelized into a solid volumetric representation with any kind of interior materials, such as rainbow, marble, wood, translucent jade, and so on.

The algorithms introduced in this paper are easy to implement and convenient to integrate into many applications, such as volume modeling, collision detection, medical simulation, volume animation, and computer art. The experimental results and data analysis for the complex objects demonstrate the effectiveness, flexibility, and diversity of this approach. Additionally, using this approach, the multi-valued volume can be efficiently created only once during the voxelization. This is much different from prior inefficient multi-valued volume computation methods, usually first generating binary volume and then applying the value to each pixel.

Another worth mention, the algorithm described in this chapter is slice-independent voxelization processing, which is critical for parallelization. Chapter 4 and 5 will present

one new parallelization scheme based on this feature without changing the sequential code to improve the total performance.

This chapter is organized as follows: Section 3.1 introduces the definition of solid object. Section 3.2 describes the definition and design of dynamic solid slice functions. Section 3.3 discusses the key principles and algorithms of dynamical solid slice creation, respectively. Section 3.4 presents real-time solid voxelization with arbitrary orientation and several GPU-based acceleration techniques. Section 3.5 provides detailed experimental results, performance and data analysis, and comparison with the existing method. The advantages of this approach are summarized in the end.

3.1 Solid Object

In order to more clearly explain solid voxelization in this research, solid object is classified into two categories, solid geometric object and solid volumetric object. From the view of solid voxelization, solid geometric object can be defined in different formulas related to solid volumetric object. The conceptual solid voxelization is defined in this section.

3.1.1 Solid Volumetric Object

Conceptually, a volumetric object ***V*** is a set of voxels in three-dimensional (3D) grid space with the size of *Nx, Ny, Nz*. It can be generally described as

$$V(x, y, z) = \{v(x, y, z) \in R^3\} \tag{3.1}$$

Where $x = 1,..., Nx, y = 1,..., Ny, z = 1,..., Nz$. The *v(x, y, z)* could be an intensity, a binary or RGB, or RGBA value, etc.

The Figure 3.1 shows a slice of a solid volumetric object containing both voxelized surface boundary and interior voxels.

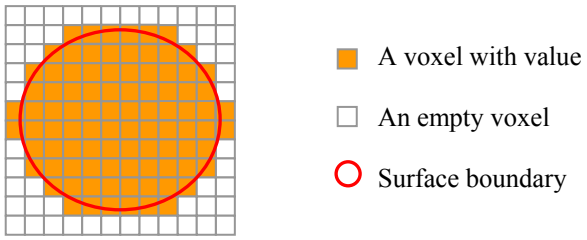


Figure 3.1: A slice of a solid volumetric object

3.1.2 Solid Geometric Object

Solid geometric objects are normally represented by their boundary surfaces without explicit interior information. The voxelization of solids is much harder than the voxelization of surface-based geometric objects because this process would require an inside test for each voxel in the volume space. For example, a binary solid might be represented by an *inside-outside function f* where *f(p)* = 1 for all points *p* inside the solid and *f(p)* = 0 elsewhere. The following formula describes such a binary solid geometric object.

$$f(p) = \begin{cases} 1 & (p \text{ inside the solid}) \\ 0 & (p \text{ outside the solid}) \end{cases} \tag{3.2}$$

In general, a boundary surface of a solid geometric object can be described as the following two cases:

i. One case is that a boundary surface of a solid is directly described by a set of polygons. Each polygon consists of a set of vertices in 3D space.

ii. The other case is that a boundary surface of a solid is directly described by an analytical object *f(x, y, z),* which represents any mathematic function to describe the surface of the solid geometric object. It could be discrete or continuous in 3D space. In computer graphics, however, any continuous surface described by a certain function *f(x, y, z)* should be sampled into a set of vertices or a set of polygons for approximate surface representation in discrete mode for display on the screen.

Thus, a solid geometric object consists of a closed surface and interior information. From the perspective of solid voxelization, it can be conceptually described as

$$O = \begin{cases} Surface & = \{p_i, i = 1,..., n\} \\ Interior & = I(x, y, z) \end{cases} \tag{3.3}$$

Where I is an interior function, which could be constant value (e.g., color and intensity) as well as a dynamic function.

3.1.3 Definition of Solid Voxelization

When a solid geometric object is converted into a volumetric object, the interior voxels should be generated correspondingly. This process is called *solid voxelization*. Unlike surface voxelization, in which only the surface boundary needs to be voxelized without any interior voxels. While solid voxelization not only needs to voxelize the same surface but also needs to fill all interior voxels. Obviously, solid voxelization is more complex in volumetric modeling and more expensive on time spent.

3.2 Solid Slice Function

Since the core idea for real-time solid voxelization is based on slice voxelization, the basic concepts related to the slices are presented in this section. Solid slice and solid slice function are defined first. Several typical solid slice functions used for this dissertation research are described as well.

3.2.1 Definition of Solid Slice

For slice-based voxelization, a volumetric object voxelized by a solid geometric object is composed of a set of solid slices. A *solid slice* is a rectangle image with interior pixel information including the boundary pixel information. Thus, the *i*th solid slice $\boldsymbol{S_i}$ can be described as

$$S_i(x, y) = \{ f(x, y, i) \in R^3 \} \tag{3.4}$$

Where $x = 1,..., M$, $y = 1,..., N$, $i = 1,..., K$, M is the width of the slice, N is the height of the slice, and K is the total number of the slices in the volume.

Besides the definition based on voxels introduced in Section 3.1.1, a volumetric object $\boldsymbol{V}$ can also be described based on slices as follows:

$$V(z) = \{S_i, i = 1,..., K\} \tag{3.5}$$

Each solid slice defined in this dissertation not only can be filled by constant values but also can be described by a certain function. Furthermore, each solid slice can be applied by different functions to generate special effects for the volume object.

3.2.2 Definition of Solid Slice Function

In theory, a solid slice function can be any function, which is static or dynamic, continuous or discontinuous in 2D or 3D space or even higher dimensional space. For example, a solid slice function could be a color, intensity, index, fixed texture, noise, procedural texture [Ebe03], filter, blending function, transformation, deformation, logic, etc. In addition, for special purpose or application needs, these functions can be defined by users themselves. Specifically, the solid slice function f_s can be described as

$$f_s(x,y,z) = \begin{cases} fixed-value \\ f(x,y) \\ f(x,y,z) \\ f_{customized}(x,y,z,...) \end{cases} \tag{3.6}$$

The design of a solid slice function constitutes an important process of real-time solid voxelization. One or more parameters can be defined in each function, or even be modified dynamically by the users during interaction since the entire solid voxelization can be done in real time.

3.2.3 Several Typical Solid Slice Functions

Several typical functions used for this dissertation are described with corresponding math functions in Table 3.2.1.

Table 3.2.1: Math function descriptions of some typical solid slice functions

Function Type	Math Function Description	Comment
FUNC_SINGLE_COLOR	$f_s(x, y, z) = color$	*color*: a constant
FUNC_COLOR_RAMP	$f_s(x, y, z) = Colorramp(color[2], z)$	*color[2]*: two constants
FUNC_INTERPOLATED_COLORS	$f_s(x, y, z) = Interpolate(color[4], x, y)$	*color[4]*: four constants
FUNC_COLORFUL_CHECKER	$f_s(x, y, z) = Chec\,\mathrm{ker}(color[2], z)$	*color[2]*: two constants
FUNC_NOISE_WOOD	$f_s(x, y, z) = Perlin_noise(x, y, z)$	Perlin noise in 3D space
FUNC_NOISE_MARBLE	$f_s(x, y, z) = Perlin_noise(x, y, z)$	Perlin noise in 3D space
FUNC_NOISE_ROCK	$f_s(x, y, z) = Perlin_noise(x, y, z)$	Perlin noise in 3D space
FUNC_MY_TYPE*	$f_s(x, y, z) = f_{\text{Customized}}(x, y, z)$	Customized functions

FUNC_SINGLE_COLOR is a fixed-value solid slice function. FUNC_COLOR_RAMP is a dynamic solid slice function with rainbow effects (i.e. each slice has different color). FUNC_INTERPOLATED_COLORS is a solid slice function, in which all the slice colors are interpolated smoothly by four input colors of four vertices on each slice. FUNC_COLORFUL_CHECKER is a solid slice function defined by one checker texture with rotation operation in a regular manner, simulating a 3D checker. FUNC_NOISE_WOOD, FUNC_NOISE_MARBLE, and FUNC_NOISE_ROCK are solid slice functions defined by Perlin noise [Per85, Per02] in 3D continuous space. FUNC_MY_TYPE* are solid slice functions, such as a semi-translucent jade effect, defined by the users themselves to meet their requirements according to their special applications.

In addition, the experimental results of some example effects of these solid slice functions are shown in Section 3.5.

3.3 Solid Slice Creation

This section describes one important masking technique. The significant mechanism of dynamic solid slice function for real-time solid slice voxelization proposed in this dissertation is also inspired by this masking technique. In this section, the key principle and algorithm of dynamic solid slice creation are discussed in detail, respectively.

To achieve real-time solid voxelization, many GPU buffers and rasterization techniques are used for solid slice creation. The core idea of this approach is to use depth buffer and stencil buffer to create a mask by surface parity check and then use GPU rasterization to create each solid slice in combination with diverse solid slice functions.

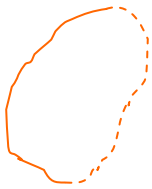
3.3.1 Surface Parity Check and Mask Creation

The surface of an object can be categorized as front or back facing, in relation to the position of the viewer. *Surface parity* refers to whether a surface in the depth buffer is inside or outside of a given volume. The parity of each depth buffer element can be determined by toggling a parity flag (i.e. front or back facing) at that pixel in front of the depth buffer. Regions of odd parity, where the flag is set to one, correspond to depth buffer elements volumetrically inside an object. The parity flag information is recorded correspondingly in the stencil buffer as a *mask*.

3.3.1.1 Front Facing and Back Facing

In computer graphics, the normal is usually used to determine which of the facets are front-facing and back-facing. By default, counterclockwise polygons are taken to be front-facing in OpenGL. In addition, it is also quite easy to use Figure 3.2 to intuitively

determine which face is front or back if polygon is a hole or not when the stencil mask is applied in the example as shown in Figure 3.3.



POLYGON	OUTSIDE	HOLE
Dashed line	Frontface	Backface
Solid line	Backface	Frontface

Figure 3.2: Determination for frontface and backface

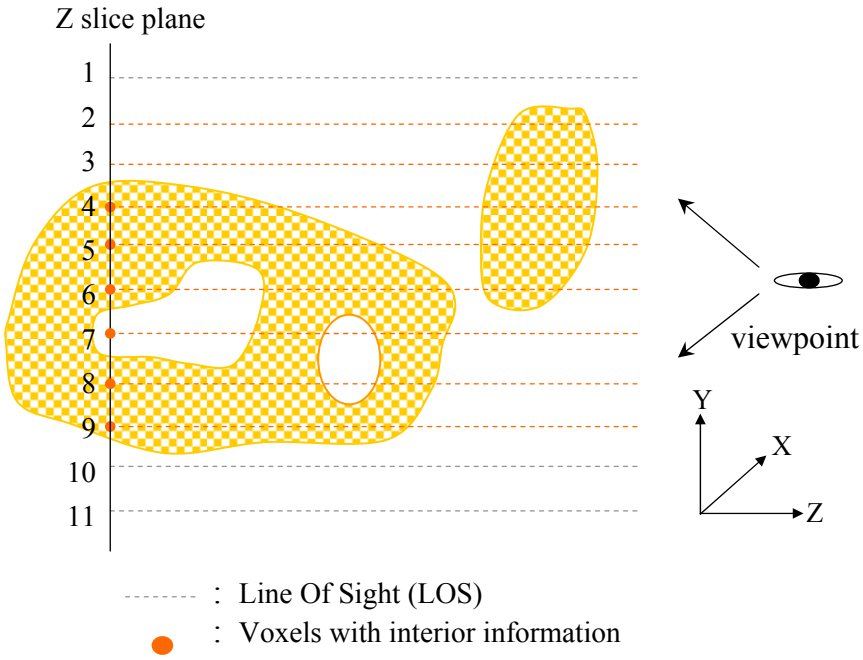


Figure 3.3: Surface parity check and mask creating

3.3.1.2 Surface Parity Checking

Figure 3.3 illustrates the principle of surface parity check on each solid slice. In the figure, the solid geometric object is clipped by a plane perpendicular to the Z-axis, called a *slice plane*. Such a slice plane is very similar to that as shown in [CF00][FL00]. The

position of the slice plane is that of the corresponding solid slice. Note that each Line Of Sight (LOS) to each pixel on the slice plane must be parallel to Z-axis in the algorithm. The parity flag is initially set to be zero. Given a slice plane, the parity of each depth buffer element is toggled at that pixel in front of the depth buffer. Along the LOS from the viewpoint, the flag is toggled when the LOS intersects with front or back facets of the solid object which are in front of the slice plane. If the final parity flag is set to one, it shows that pixel is inside the solid object. Otherwise, the pixel is outside the solid object.

Table 3.3.1: An example of surface parity check

LOS	Parity Flag Toggling	Stencil
1	**0**	**0**
2	01**0**	**0**
3	01**0**	**0**
4	010**1**	**1**
5	010**1**	**1**
6	01010**1**	**1**
7	0101**0**	**0**
8	010**1**	**1**
9	010**1**	**1**
10	0**1**	**1**
11	**0**	**0**

Table 3.3.1 gives a detailed example of surface parity check in relation to the Figure 3.3. It describes how the parities of different depth buffer elements are toggled through their corresponding LOSs step by step. In the table, the parity flag with 1 means the LOS

just crosses the front facet of the solid object. In contrast, the parity flag with 0 means the LOS passes through the back facet. The total number of 1's toggled is the count of the front facing as shown in the LOS 4, 5, 6, 8, 9, and 10. Similarly, except first initial 0, the total number of 0's is the count of the back facing as shown the LOS 1, 2, 3, 7, and 11.

Additionally, there are important constraints on surfaces used for parity testing. Parity logic depends on no surface being interior or exterior to the volume it represents. Also, all boundaries of the volume should be covered by surface. This means that the surface should be closed, and should not contain any holes caused by an incomplete description, such as missing polygons. Additionally, the surface must not self-intersect, forming folds or loops.

3.3.1.3 Creating a Mask

In this approach, stencil buffer is used as a mask to store the parity check, which will be used to determine the interior information of the clipped solid object. At the same time of surface parity checking, the parity flag information is recorded correspondingly in the stencil buffer as a *mask*. In Table 3.3.1, the final stencil value is the same as the last value of the parity of the depth buffer element.

After the mask is created in the stencil buffer, the solid slice can be obtained through the mask in combination with the solid slice function. The rest of the sections in this chapter will give more details about creating a solid slice.

In general, all the processes of the surface parity check and mask creating can be implemented by GPU hardware. This important feature makes full use of GPU hardware acceleration. Consequently, the solid slice can be obtained at interactive or even real-time speed, which only depends on the complexity of input solid slice functions.

3.3.2 Principle of Solid Slice Creating

The principle of one solid slice creation based on masking and solid slice function techniques is illustrated in Figure 3.4. Note that, in this approach, since the final framebuffer will be regarded as one slice, the projection needs to be set to the parallel projection to avoid any distortion in the slice plane.

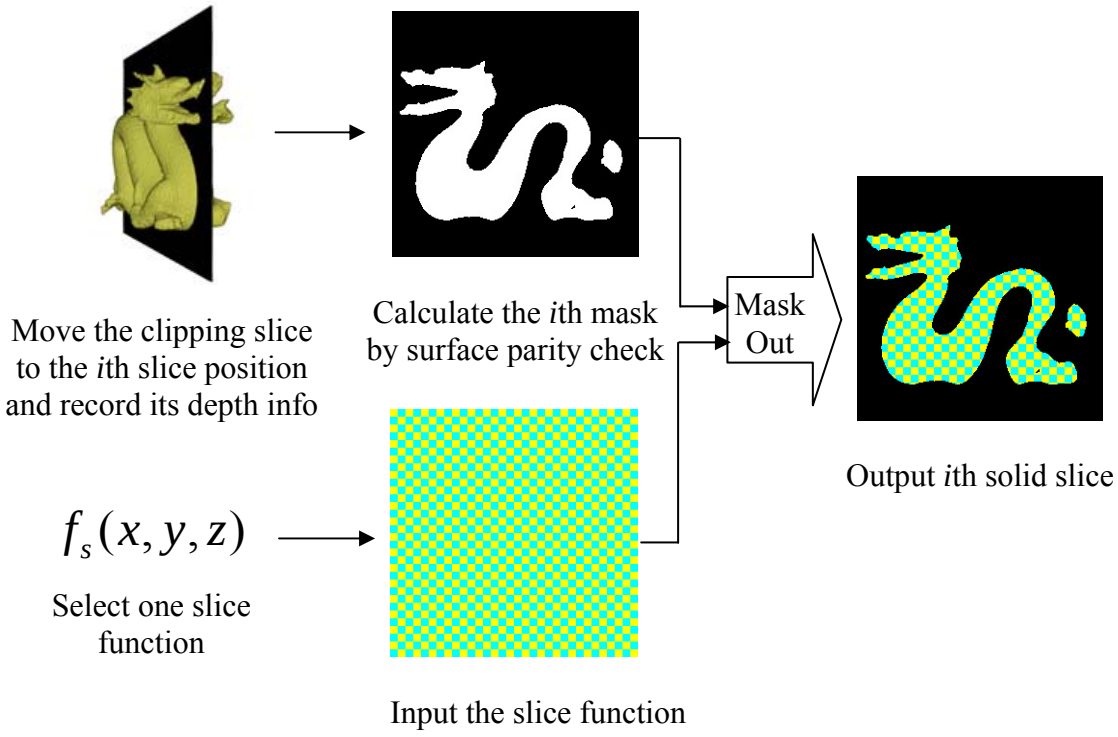


Figure 3.4: Principle of one solid slice creating

More specifically, the basic idea to create a solid slice by one specified solid slice function is described as follows:

- Move one slice plane to a certain position in parallel with the Z-axis in 3D space.

- Set the depth buffer to be the same size of the slice, and record the depth information with the position of the slice plane.
- Check the surface parity of the input solid geometric object.
- Use a stencil buffer with the same size to generate a mask of the solid object clipped by the slice plane at that slice position.
- Select one slice function and apply it to the solid object.
- Use the stencil mask to mask out the interior features of the clipped solid object into the framebuffer.
- Output the framebuffer as one solid slice.

Thus, one solid slice is created with interior information specified by the slice function.

3.3.3 Algorithm of Solid Slice Creating

The algorithm of solid slice creating is summarized in Figure 3.5. In the figure, the input parameters are the display list of the geometric object, L, the size of the solid slice, Nx and Ny, the z position of the slice plane, and a solid slice function, f_s, specified by the user. The output is only the created solid slice, composed of Nx by Ny pixels/voxels. In fact, the outcome of the framebuffer is the solid slice just created as need.

Algorithm 3.1: Solid Slice Creating

CreateSolidSlice()

Input:

$L \leftarrow$ a display list of a closed surface-based geometric object

$z \leftarrow$ the z position of the solid slice

$Nx, Ny \leftarrow$ the size of the solid slice

$f_s \leftarrow$ a specified solid slice function

Output:

$S \leftarrow$ the solid slice

Begin

1 Clear the color, stencil, and depth buffers

2 Set the depth buffer using the slice position with the size of Nx by Ny

3 Call the display list L to draw the geometry object

4 Check surface parity

5 Apply the solid slice function f_s to the slice in the color buffer

6 Use stencil mask to mask out the solid slice

6.1 If stencil buffer value is 1 **then**

Corresponding portion of ith slice is updated by slice function in color buffer

6.2 Else cleared to background color

End

7 Get final voxelized solid slice S from the framebuffer

8 Output the entire voxelized solid slice S

End

Figure 3.5: Algorithm of solid slice creating

3.3.4 Advantages of Solid Slice Creating Algorithm

There are several advantages in this solid slice creating approach as follows:

- A surface parity check is used for faster masking creation instead of counting the total number of front and back faces.
- A solid geometric object is drawn only once instead of drawing it twice to determine front and back faces.
- Dynamic solid slice functions are employed to create diverse solid slices instead of a single color or binary solid slice.
- The algorithm can be fully implemented on the standard graphics system and accelerated by graphics hardware.
- Only one depth buffer, one stencil buffer, and one color buffer are utilized to generate one solid slice at one time. If multiple or larger buffers are used, more solid slices can be created independently at the same time. The total performance can be improved accordingly.
- Slice-independent processing is critical for parallelization. The whole process for creating each solid slice is independent to other slices in the volume. This important feature is suitable for many parallel processing methods potentially. Chapter 4 and Chapter 5 will discuss a new multi-core pipelined parallelization method using this feature to improve the total performance.
- Each solid slice is generated by hardware-accelerated 2D scan conversion in this approach. Since 3D scan conversion is an extension of 2D scan conversion in the 3D space, the entire solid volumetric object composing of a stack of solid slices is actually generated by hardware-accelerated 3D scan

conversion. Section 3.4 will discuss more details for this hardware-accelerated 3D scan conversion approach.

3.4 GPU-based Real-time Solid Voxelization

Real-time solid voxelization is the goal of this research. In this section, the algorithm of slice-independent real-time solid voxelization is first described in detail. This approach significantly improves solid voxelization speed in real time as well as creates diverse multi-valued solid volumetric models with various kinds of slice functions. In addition, several key real-time acceleration techniques are introduced.

3.4.1 Algorithm of Real-time Solid Voxelization

The solid voxelization algorithm proceeds by moving a slice plane, which is parallel to the projection plane, with a constant step size in a front-to-back or back-to-front order in volume space. More specifically, the major steps of the algorithms are described as follows:

- First, one solid slice function is selected and initialized or preprocessed.
- Second, some transformations will be applied to the solid geometric object in order to make its voxelization orientation consistent with Z-axis. In other words, this algorithm can do voxelization in an arbitrary orientation.
- Third, each slice in the volume space is processed by looping with the number of total slices. For each new slice, if it is inside the bounding box of the solid geometric object, a new solid slice will be created by calling *CreateSolidSlice(),* the solid slice creating algorithm as described in Figure

3.5. Then it will be directly stored into a given 3D texture volume without taking time to do swapping between GPU and main memory.

- Output of 3D texture volume.

The algorithm of real-time solid voxelization for arbitrary orientation is summarized in Figure 3.6.

Algorithm 3.2: Real-time Solid Voxelization

SolidVoxelization()

Input:

$O \leftarrow$ a closed surface-based object

$Nx, Ny, Nz \leftarrow$ size of the output volume

$Bmin, Bmax \leftarrow$ minimum and maximum 3D vectors (x, y, z) of the bounding box

$f_s \leftarrow$ a specified solid slice function

Output:

$V \leftarrow$ the voxelized solid volume

Begin

1 Set the viewport (0, 0, *Nx, Ny*)

2 Set orthogonal projection

3 Set background color to be (0, 0, 0, 0)

4 Do transformation for the object, *O,* to make its orientation specified for voxelization consistent with *Z* axis. Generate the display list *L* for the geometric object *O* drawing.

5 Initialize or preprocess the slice function f_s

6 For each slice *i = 0* to *Nz – 1* **do**

6.1 If slice *i* insides the bounding box *(Bmin, Bmax)* **then**

6.1.1 *i*th solidSlice ← CreateSolidSlice(*L*, *z(i)*, f_s)

6.1.2 Insert *i*th solidSlice into the 3D texture volume *V*

End

End

7 Output the entire volume *V*

End

Figure 3.6: Algorithm of real-time solid voxelization

3.4.2 Real-time Acceleration Techniques

The entire real-time solid voxelization employs many hardware (GPU) and software acceleration techniques. Most operations and computations are carried out on GPU thus avoiding unnecessary data transfer. Therefore, the total solid voxelization can be implemented in real time.

Several key techniques are used for this algorithm summarized as follows:

♦ **Z-bucket**: In fact, Z-bucket algorithm only simply uses the bounding boxes to exclude outside objects when the scene is drawn.

♦ **Preprocessing**: Some slice functions, such as `FUNC_COLOR_RAMP` and `FUNC_NOISE_*`, are pre-computed or preprocessed as much as possible. Then the preprocessing results are put into the memory for later use.

♦ **Display list**: The OpenGL display list technique is used for fast geometric object drawing during the solid slice creation.

♦ **OpenGL smooth shading**: OpenGL smooth shading is used here for color fast interpolation, such as slice function `FUNC_INTERPOLATED_COLORS`.

♦ **OpenGL buffers**: Just as mentioned before, the key algorithm of solid slice creation makes full use of OpenGL buffer (depth buffer, stencil buffer, color buffer, etc.) techniques and can be accelerated by graphics hardware.

♦ **3D Texturing**: 3D texturing is much faster than a stack of 2D texturing without time-consuming swapping between main memory and GPU memory.

♦ **Programmable GPU**: It is very flexible to use programmable GPU [Fer04, Pha05] to implement some special processing or effects pixel by pixel in 2D texturing or voxel by voxel in 3D texturing through by fragment programs.

3.5 Experimental Analysis

3.5.1 Experimental Results

Some experimental image results of the new algorithm of real-time solid voxelization are presented in Figure 3.7, Figure 3.8, Figure 3.9, Figure 3.10, and Figure 3.11. The time spent on creating volumetric models in different resolutions and sizes are listed in Table 3.5.1, Table 3.5.2, Table 3.5.3, Table 3.5.4, and

Table 3.5.5. The performance and analysis including the comparison with other methods are described in Section 3.5.2 and Section 3.5.3. The corresponding statistic graphs are shown in Figure 3.13 to Figure 3.20.

All tests were run on a Dell PC with Pentium (R) D a single processor 3.0GHZ and 2GB RAM equipped with an Nvidia GeForce GTX 260 (8-bit stencil buffer, 24-bit depth buffer, 896MB texture memory, and 3D texture mapping supported). The program is written in Microsoft Visual Studio 2005 (VC++ 8.0 /.NET 2.0), OpenGL 2.0, and Cg 2.0. All volumetric data models were rendered using 3D texture mapping.

In Figure 3.7, (a) is original polygonal dragon models by surface-based rendering. (b) and (d) are the solid voxelization results by using the solid slice function `FUNC_COLOR_RAMP` defined in Table 3.2.1. However, the voxelization orientation and color settings are different during the voxelization initialization. The orientation of (b) is from the dragon head to its tail. (d) is from the left side to the right side. (c) is the solid

voxelization result by using the slice function `FUNC_COLORFUL_CHECKER`. (f) and (h) are solid voxelization results by using the slice function `FUNC_NOISE_MARBLE` and `FUNC_NOISE_WOOD`. (e), (g), (i) are the results after (d), (f), and (h) are cut in one block, respectively. All interior materials can be seen in these image results. It shows that (g) and (i) keep continuous in 3D space by using Perlin 3D noise during solid voxelization.

In Figure 3.8, (a), (b), (c), and (d) are higher resolution image results of some examples of solid voxelization by the size of 256^3 in Figure 3.7. Much more details are shown in these images.

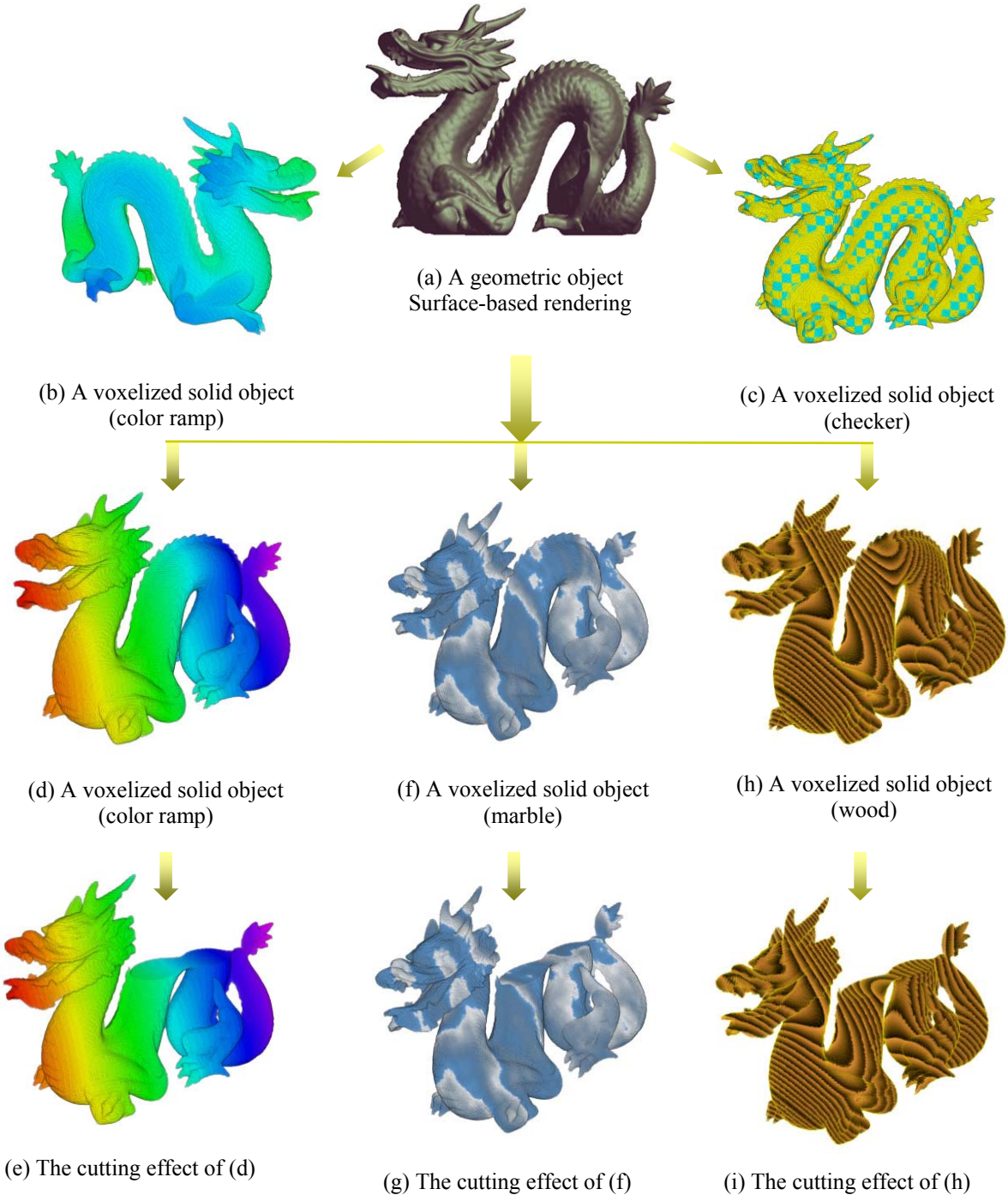
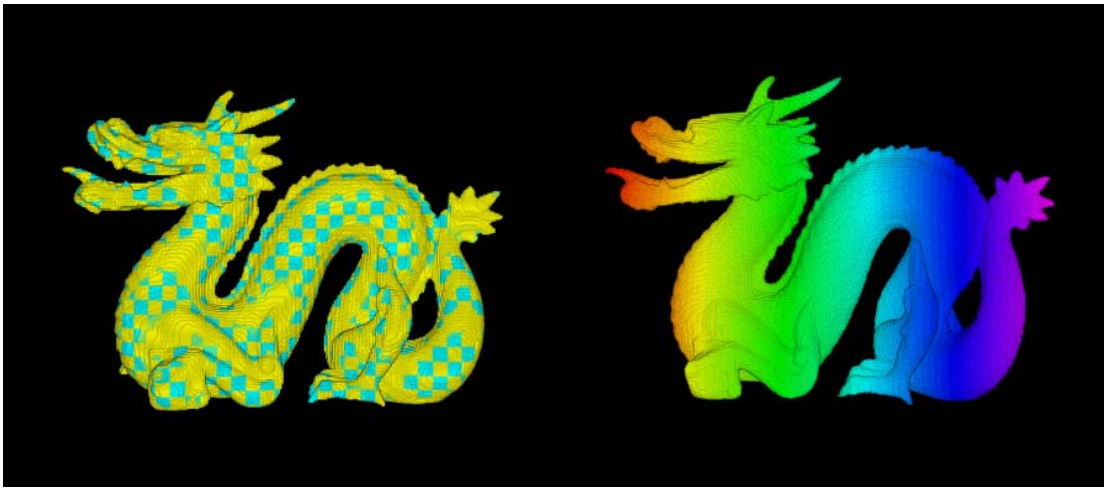
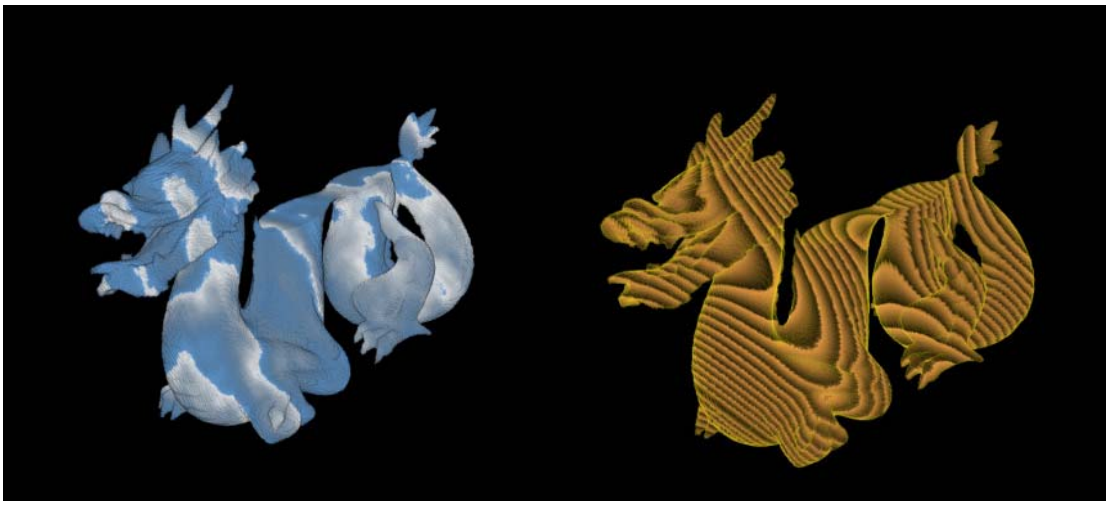


Figure 3.7: Solid voxelization examples of the Dragon geometric model



(a) A voxelized solid object (b) A voxelized solid object



(c) A cut marble solid volume (d) A cut wood solid volume

Figure 3.8: Solid voxelization results in 256^3 of the Dragon geometric model

Likewise, in Figure 3.9, (a) is original polygonal Buddha models by surface-based rendering. (b) is the solid voxelization result by using the solid slice function `FUNC_COLOR_RAMP` defined in Table 3.2.1. (c) is the solid voxelization result by using the slice function `FUNC_INTERPOLATED_COLORS`. (d) and (f) are solid voxelization results by using the slice function `FUNC_NOISE_MARBLE` and `FUNC_NOISE_ROCK`. (e) and (g) are the cutting effects of (d) and (f), respectively. All interior materials can be seen in these image results. It shows that (e) and (g) keep continuous in 3D space by using Perlin 3D noise during solid voxelization. (h) is the solid voxelization result of the customized solid slice function to simulate the effect of semi-translucent jade. (i) is the back effect of such a volumetric Buddha.

In Figure 3.10, the image results from (a) to (g) are generated in higher resolution of some examples of solid voxelization by the size of 256^3 in Figure 3.9. Much more details are shown in these images.

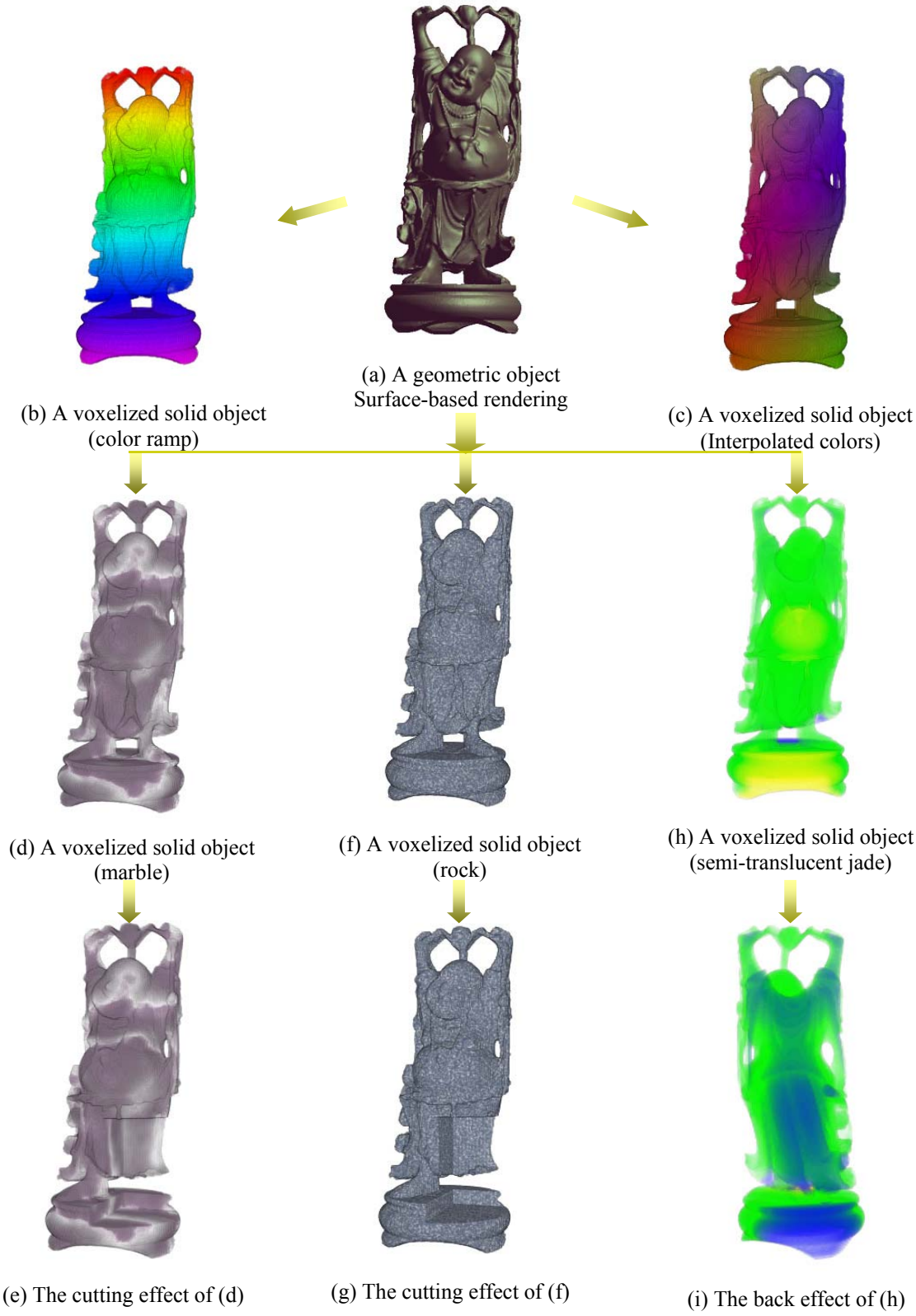
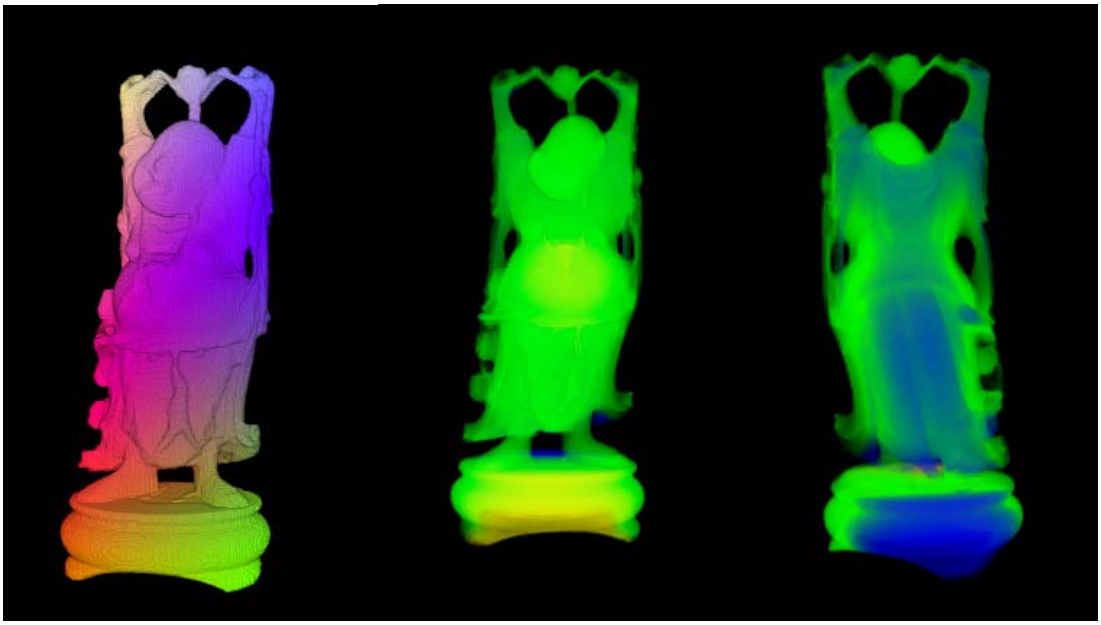


Figure 3.9: Solid voxelization examples of the Buddha geometric model



(a) A voxelized colorful solid (b) and (c) A voxelized semi-translucent solid

(e) A voxelized rock solid (f) A voxelized marble solid (g) A cut marble solid volume

Figure 3.10: Solid voxelization results in 256^3 of the Buddha geometric model

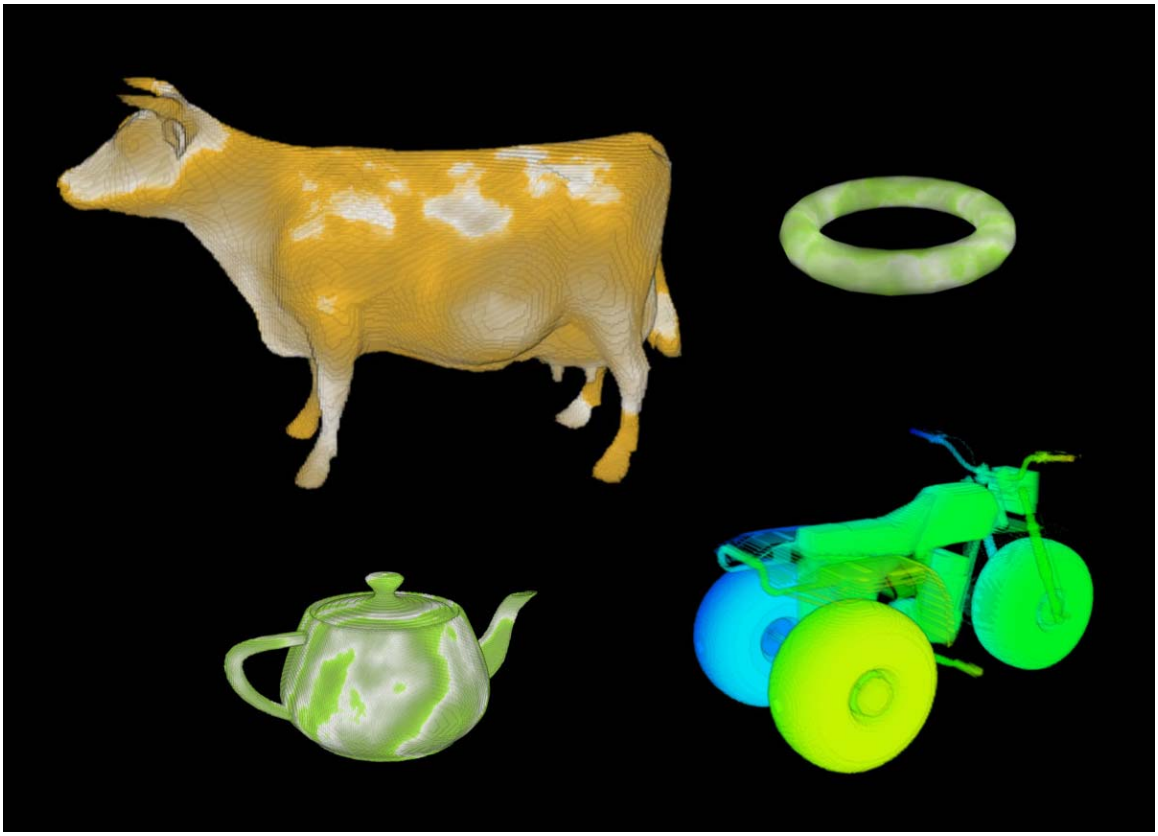


Figure 3.11: Solid voxelization results in 256^3 of the other models (cow, torus, teapot, ATC)

Additionally, Figure 3.11 shows the solid voxelization results of the some other models, such as cow, torus, teapot, ATC, etc., using different slice functions by the size of 256^3.

3.5.2 Performance

This section will show the time spent on solid voxelization of different models using different slice functions, such as single-color-based, color-based, and texture-based functions. In addition, for comparison, some other existing solid voxelization methods are implemented with the same geometric models on the same development platform.

Just as mentioned in Section 2.4, most of existing solid voxelization approaches still uses pure-software methods, such as flood-filling or 3D scan conversion, to fill the interior voxels. These algorithms are extremely slow. Only a few solid voxelization algorithms by hardware acceleration, such as Fang and Chen's XOR-based method [Che98, CF00] using very thin projection to catch the slices, can reach interactive speed in some conditions. To compare with this new approach, one existing method of fast XOR-based solid voxelization by hardware acceleration [CF00][DL00] has been implemented in the same conditions.

3.5.2.1 Generated volumetric Models Compared with Other Method

Although the XOR-based method has improved the solid voxelization speed at interactive time, it only generates the binary volumetric model. Additionally, it produces erroneous prolonged lines due to the XOR operation as shown in Figure 3.12.

In the figure, these XOR-based solid voxelization results in 128^3 are generated using the same models, such as dragon, Happy Buddha, cow, ATC, etc., as the new approach does. Some missing voxels are shown in the front side of the dragon model in the up-left image, the corresponding prolonged lines are shown in the back side of the dragon model

in the up-right image; likewise, the erroneous prolonged lines are generated in cow, buddha, and ATC models.

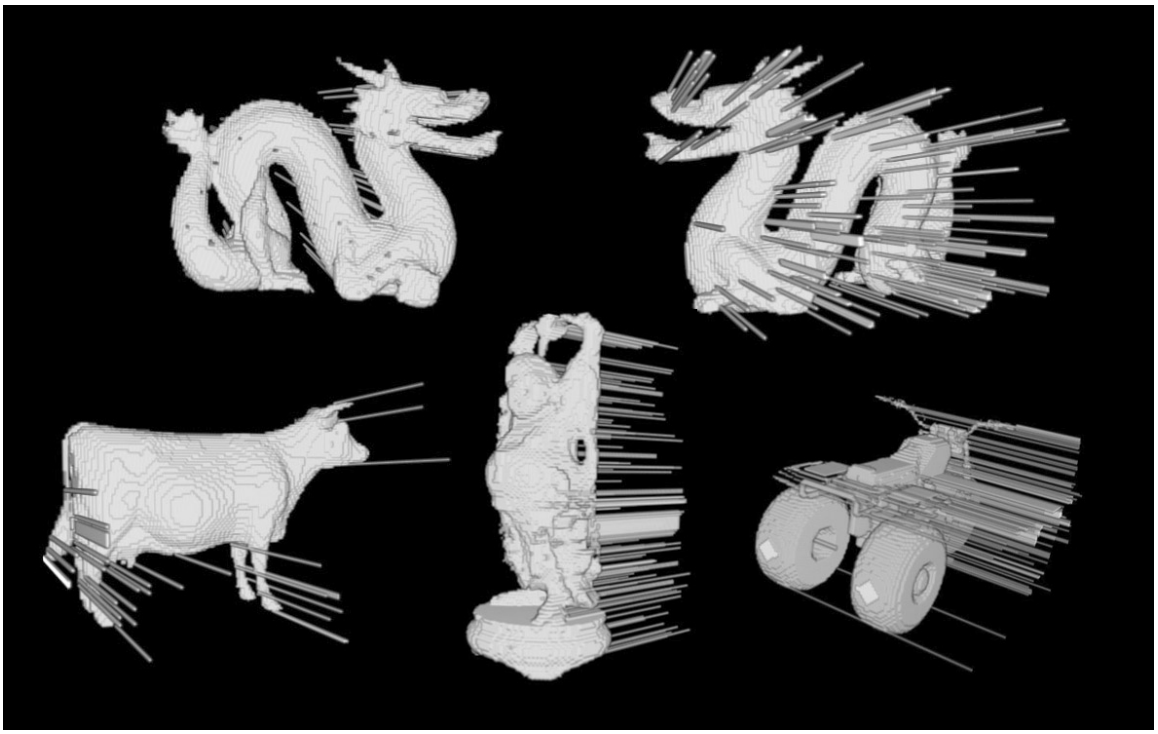


Figure 3.12: XOR-based Solid voxelization results in 128^3 of some models

(Some missing voxels are shown in the front side of the dragon model in the up-left image, the corresponding prolonged lines are shown in the back side of the dragon model in the up-right image; likewise, the erroneous prolonged lines are generated in cow, Buddha, and ATC models.)

In comparison with the XOR-based method, the new approach described in my dissertation can not only increase the performance at least two times as shown in the tables and figures of the following sections (such as Figure 3.20), but also generate multi-valued volumetric models. Besides the principles of two methods are different, another key difference is that the new approach only set the entire projection once at the

beginning while a very thin projection has to be set for each slice for XOR-based method. This method costs extra more time for projection calculation, which makes the performance lower than ours. Furthermore, the new approach significantly eliminates the erroneous prolonged lines and makes this approach feasible in both theory and practice.

3.5.2.2 Time and Speed Comparison with Other Method

All the experimental voxelization time in this dissertation consists of two parts. One is exact voxelization time. The other is the time of slice texture copied into 3D volume buffer.

In Table 3.5.1 (a), different resolutions of the dragon models were voxelized using `FUNC_SINGLE_COLOR` slice function in solid model by the size of 64^3, 128^3, 256^3, and 512^3, respectively. Additionally, the time spent on the XOR-based solid voxelization method with the corresponding models and sizes is also shown in Table 3.5.1 (a). Table 3.5.1 (b) describes the ratio of the solid voxelization speed using the new approach compared to the XOR-based method. In this table, it clearly shows that the performance can be improved at least two times or so using the new approach.

Table 3.5.1: Time and speed comparison of solid voxelization of the Dragon models

Model	# of Vertices	# of Polygons	Solid Voxelization Time (ms)							
			64^3 (1MB)		128^3 (8MB)		256^3 (64MB)		512^3 (512MB)	
			XOR-based	New Alg.	XOR-based	New Alg.	XOR-based	New Alg.	XOR-based	New Alg.
Dragon _res4	5,205	**11,102**	256	109	487	205	1105	423	3917	1055
Dragon _res3	22,998	**47,794**	964	537	1726	890	3774	1671	9092	3334
Dragon _res2	100,250	**202,520**	4271	1877	7909	3450	15099	6694	30962	13718
Dragon	437,645	**871,414**	19997	10919	37486	18217	70469	34577	139807	64438

(a) Solid voxelization using `FUNC_SINGLE_COLOR` slice function in comparison to the XOR-based solid voxelization method

Model	# of Vertices	# of Polygons	Ratio of Solid Voxelization Time			
			64^3 (1MB)	128^3 (8MB)	256^3 (64MB)	512^3 (512MB)
Dragon_res4	5,205	**11,102**	2.35	2.38	2.61	3.71
Dragon_res3	22,998	**47,794**	1.80	1.94	2.26	2.73
Dragon_res2	100,250	**202,520**	2.28	2.29	2.26	2.26
Dragon	437,645	**871,414**	1.83	2.06	2.04	2.17

(b) Solid voxelization Time Ratio of XOR-based algorithm to the new algorithm

Table 3.5.2: Time and speed comparison of solid voxelization of the Happy Buddha models

Model	# of Vertices	# of Polygons	Solid Voxelization Time (ms)							
			64^3 (1MB)		128^3 (8MB)		256^3 (64MB)		512^3 (512MB)	
			XOR-based	New Alg.	XOR-based	New Alg.	XOR-based	New Alg.	XOR-based	New Alg.
Buddha_res4	7,108	**15,536**	347	184	654	301	1411	504	4603	1187
Buddha _res3	32,328	**67,240**	1354	714	2482	1019	4944	1976	11306	4075
Buddha _res2	144,647	**293,232**	5661	2816	10901	5212	22152	9305	45547	18522
Happy Buddha	543,652	**1,087,716**	25645	13127	47201	21646	95201	39437	184047	73469

(a) Solid voxeliaztion using `FUNC_SINGLE_COLOR` slice function in comparison to the XOR-based solid voxelization method

Model	# of Vertices	# of Polygons	Ratio of Solid Voxelization Speed			
			64^3 (1MB)	128^3 (8MB)	256^3 (64MB)	512^3 (512MB)
Dragon_res4	5,205	**11,102**	1.89	2.17	2.80	3.88
Dragon_res3	22,998	**47,794**	1.90	2.44	2.50	2.77
Dragon_res2	100,250	**202,520**	2.01	2.09	2.38	2.46
Dragon	437,645	**871,414**	1.95	2.18	2.41	2.51

(b) Solid voxelization speed Ratio of the new algorithm to the XOR-based algorithm

In Table 3.5.2 (a), different resolutions of the Happy Buddha models were voxelized using `FUNC_SINGLE_COLOR` slice function in solid model by the size of 64^3, 128^3, 256^3, and 512^3, respectively. Additionally, the time spent on the XOR-based solid voxelization method with the corresponding models and sizes is also shown in Table 3.5.2 (a). Table 3.5.2 (b) describes the ratio of the solid voxelization speed using the new method

compared to the XOR-based method. Likewise, it clearly shows that the performance can be improved at least two times or so using the new approach.

Table 3.5.3: Time and speed comparison of solid voxelization of the other models

Model	# of Vertices	# of Polygons	Solid Voxelization Time (ms)											
			64^3 (1MB)			128^3 (8MB)			256^3 (64MB)			512^3 (512MB)		
			XOR-based	New Alg.	Ratio	XOR-based	New Alg.	Ratio	XOR-based	New Alg.	Ratio	XOR-based	New Alg.	Ratio
Cow	2,903	**3,263**	101	28	3.61	209	51	4.10	623	121	5.15	3049	387	7.88
ATC	11,577	**11,344**	260	140	1.86	511	264	1.94	1147	532	2.16	3937	1404	2.80

Apart from the time and speed data for the dragon and Buddha models with different resolutions, for the cow and ATC models, the time spent on the new solid voxelization algorithm using `FUNC_SINGLE_COLOR` slice function and the time spent on the XOR-based method are shown in Table 3.5.3. The ratio of the solid voxelization speed using the method compared to the XOR-based method is also shown in Table 3.5.3.

3.5.2.3 Time Spent on Solid Voxelization Using Other Slice Functions

Table 3.5.4 shows the time spent in voxelizing the Dragon_res4 model using some other color-based slice functions, such as `FUNC_COLOR_RAMP` and `FUNC_INTERPOLATED_COLORS`. The solid voxelization time spent is little more than those using `FUNC_SINGLE_COLOR`. These kinds of functions also include `FUNC_MY_TYPE1` (semi-translucent jade).

Table 3.5.4: Time Spent of solid voxelization using color-based slice functions

Color-based Slice Function	Solid Voxelization Time of the Dragon_res4 Model (sec)			
	64^3 (1MB)	128^3 (8MB)	256^3 (64MB)	512^3 (512 MB)
FUNC_COLOR_RAMP	0.14	0.24	0.46	1.14
FUNC_INTERPOLATED_COLORS	0.14	0.24	0.46	1.16

Table 3.5.5: Time spent on solid voxelization using texture-based slice functions

Texture-based Slice Function	Solid Voxelization Time of the Cow Model (sec)			
	64^3 (1MB)	128^3 (8MB)	256^3 (64MB)	512^3 (512MB)
FUNC_COLORFUL_CHECKER	0.04	0.12	0.68	4.66
FUNC_NOISE_WOOD	0.06	0.22	1.36	9.93
FUNC_NOISE_ROCK	0.09	0.51	3.76	29.13
FUNC_NOISE_MARBLE	0.24	1.70	13.39	106.56

(a) Solid voxelization using different texture-based slice functions for the Cow model

Texture-based Slice Function	Solid Voxelization Time of the Dragon_res4 Model (sec)			
	64^3 (1MB)	128^3 (8MB)	256^3 (64MB)	512^3 (512MB)
FUNC_COLORFUL_CHECKER	0.11	0.29	1.18	6.80
FUNC_NOISE_WOOD	0.18	0.46	2.17	14.10
FUNC_NOISE_ROCK	0.20	0.83	5.43	40.29
FUNC_NOISE_MARBLE	0.44	2.50	18.12	147.03

(b) Solid voxelization using different texture-based slice functions for the Dragon_res4 model

The time spends on solid voxelization using other texture-based slice functions for the cow and Dragon_res4 models are shown in

Table 3.5.5. Obviously, for some slice functions using 3D Perlin noise, the processing time is mainly spent on time-consuming 3D noise generation for each slice without hardware acceleration in this dissertation so far.

Likewise, all the experimental voxelization time consists of exact voxelization time and the time of slice texture copied into 3D volume buffer.

3.5.3 Analysis

3.5.3.1 Time Complexity

According to the core algorithms mentioned in Figure 4.5 and Figure 4.6, the time complexity **T** of this new solid voxelization can be described as

$$\begin{aligned} T &= n(T_{drawObj} + T_s(f_s) + T_{bufOp}) \\ &= O(nm) + nT_s(f_s) + nT_{bufOp} \end{aligned} \tag{3.7}$$

where n is the number of the slices, m is the number of polygons of the solid geometric object, $T_{drawObj}$ is the time complexity of the object drawing for one slice, T_{bufOp} is the time complexity of one-slice buffer operations (including stencil buffer, depth buffer, color buffer, etc.), $T_s(f_s)$ is the time complexity of the slice function. $T_s(f_s)$, and f_s is a slice function. $T_s(f_s)$ may be a constant or a variable depending on the solid slice function f_s. If the slice function is constant, the time complexity **T** is

$$T \approx O(nm) \tag{3.8}$$

That is to say, since the preprocessing and caching (i.e. OpenGL display list) techniques are employed for the geometric object drawing and GPU memory are employed for fast 3D volume storage, the time spent on the voxelization of the same geometric model is only related to the number of the slices, n, when the slice function $T_s(f_s)$ is a constant.

Moreover, all of buffer operations including rasterization are accelerated by the GPU. For the same object, T_{bufOp} increases very slowly with the size of the slice increasing when the number of the slices is fixed. The following experimental data results show this in Figure 3.13.

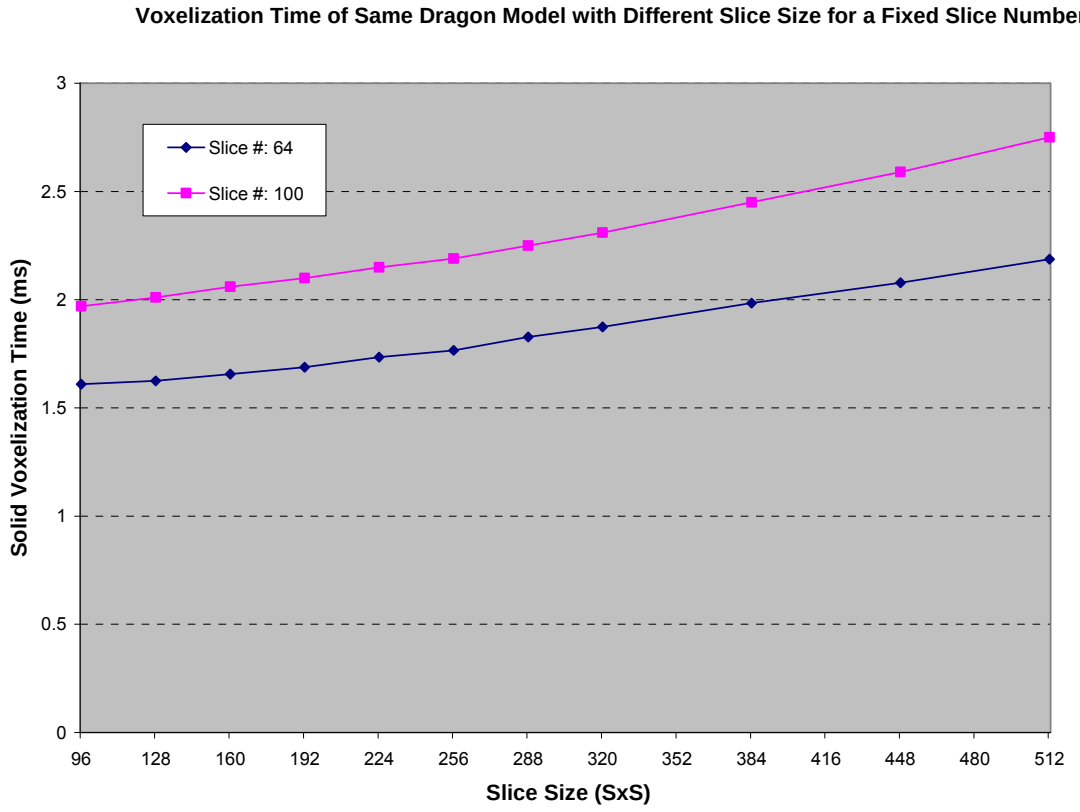


Figure 3.13: The statistic graph of solid voxelization time of Dragon_res4 model with different slice size for a fixed slice number

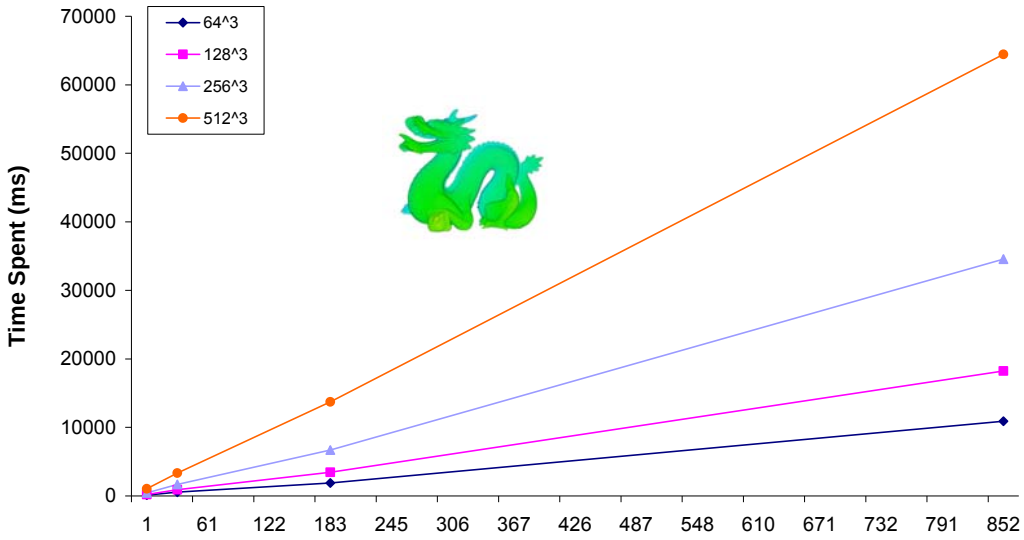
3.5.3.2 Data Analysis

In the following three statistic graphs, Figure 3.14, Figure 3.15, and Figure 3.16, $T_s(f_s)$ is a constant due to the slice function `FUNC_SINGLE_COLOR`. According to the Table 3.5.1

(a) and Table 3.5.2 (a) and, Figure 3.14 and Figure 3.15 show the statistic graphs of the solid voxelization time spent of the dragon model and Buddha model, respectively. Figure 3.16 shows the statistic graph of the solid voxelization time of all the models in Table 3.5.1, Table 3.5.2, and Table 3.5.3 with the slice function `FUNC_SINGLE_COLOR`. In all these figures, it clearly shows that no matter how many the polygons of the models there are, for the same polygonal model with the number of m, the voxelization time increases only by the slice size n times.

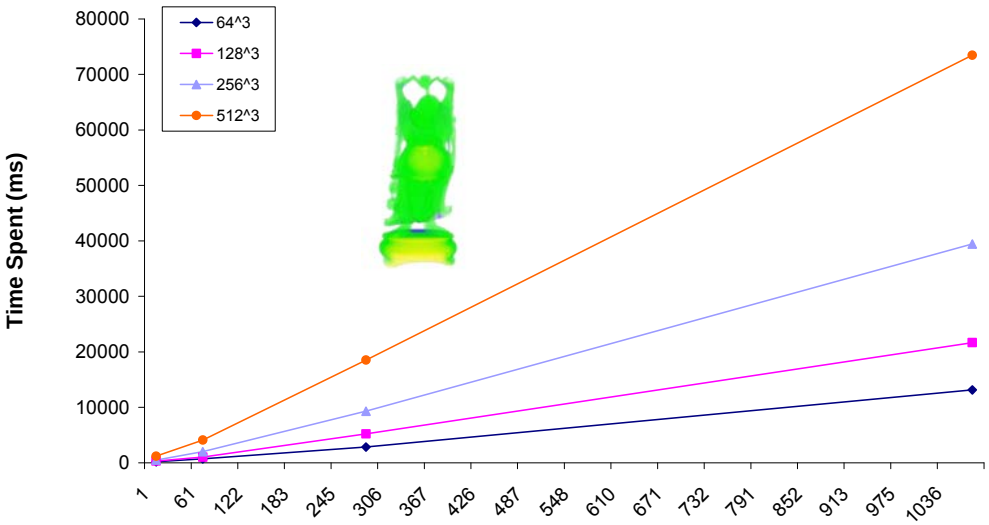
However, in Figure 3.18 and Figure 3.19, since $T_s(f_s)$ is a variable function (i.e., texture-based slice function), for the same polygonal model, the voxelization time relates to both the slice size n and the slice function f_s with non-linear increase. Figure 3.17 is the results using the same model as in Figure 3.18 but with a constant slice function. From these two figures, it clearly shows the difference of time complexity with the impact of the slice functions between them.

Hence, the voxelization time of all the experimental testing exactly follows the time complexity as shown in Equation 3.7 and Equation 3.8. All current experiments are run on Nvidia GeForce 260 with 896MB texture memory. The maximum size of the generated volumetric models in 512^3 needs 512MB texture memory. So total memory cost is within the main memory size and GPU memory size. However, once the memory cost exceeds the memory sizes, the processing time may increase a lot due to large amount of time spent in memory swapping between GPU and main memory or main memory and hard disk.



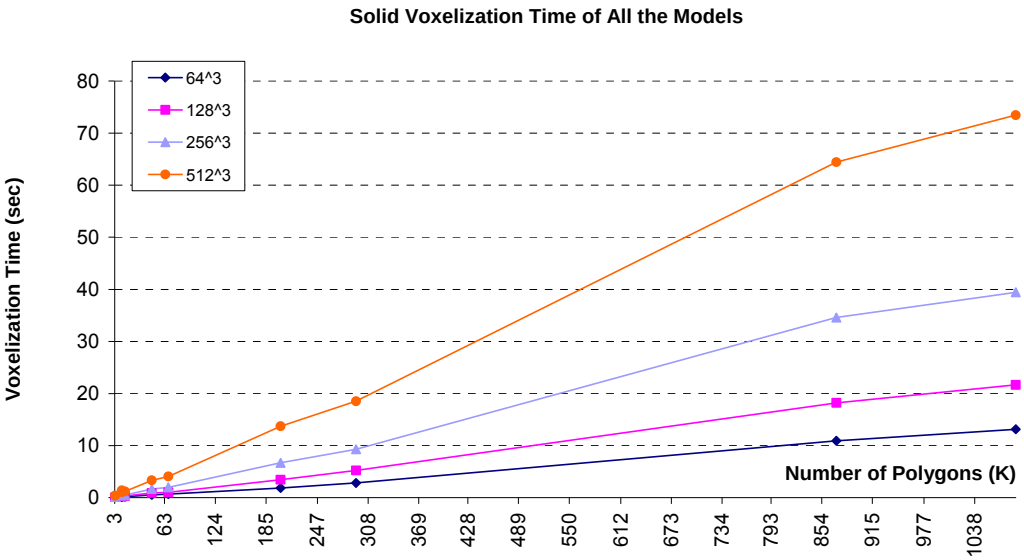
	11	48	203	871
64^3	109	537	1877	10919
128^3	205	890	3450	18217
256^3	423	1671	6694	34577
512^3	1055	3334	13718	64438

Figure 3.14: The statistic graph of solid voxelization time of Dragon models with `FUNC_SINGLE_COLOR`



	16	67	293	1088
64^3	184	714	2816	13127
128^3	301	1019	5212	21646
256^3	504	1976	9305	39437
512^3	1187	4075	18522	73469

Figure 3.15: The statistic graph of solid voxelization time of Buddha models with `FUNC_SINGLE_COLOR`



	3	11	11	16	48	67	203	293	871	1088
64^3	0.028	0.109	0.14	0.184	0.537	0.714	1.877	2.816	10.919	13.127
128^3	0.051	0.205	0.264	0.301	0.89	1.019	3.45	5.212	18.217	21.646
256^3	0.121	0.423	0.532	0.504	1.671	1.976	6.694	9.305	34.577	39.437
512^3	0.387	1.055	1.404	1.187	3.334	4.075	13.718	18.522	64.438	73.469

Figure 3.16: The statistic graph of solid voxelization time of all the models with `FUNC_SINGLE_COLOR`

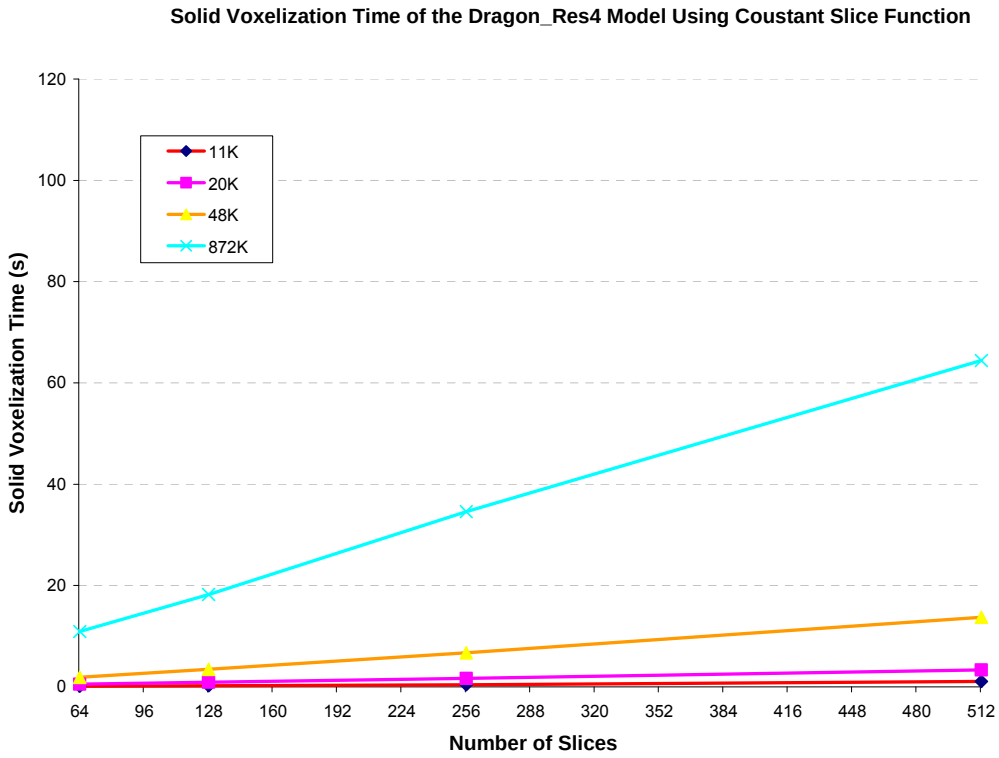


Figure 3.17: Solid voxelization time of the Dragon_res4 model using a constant slice function

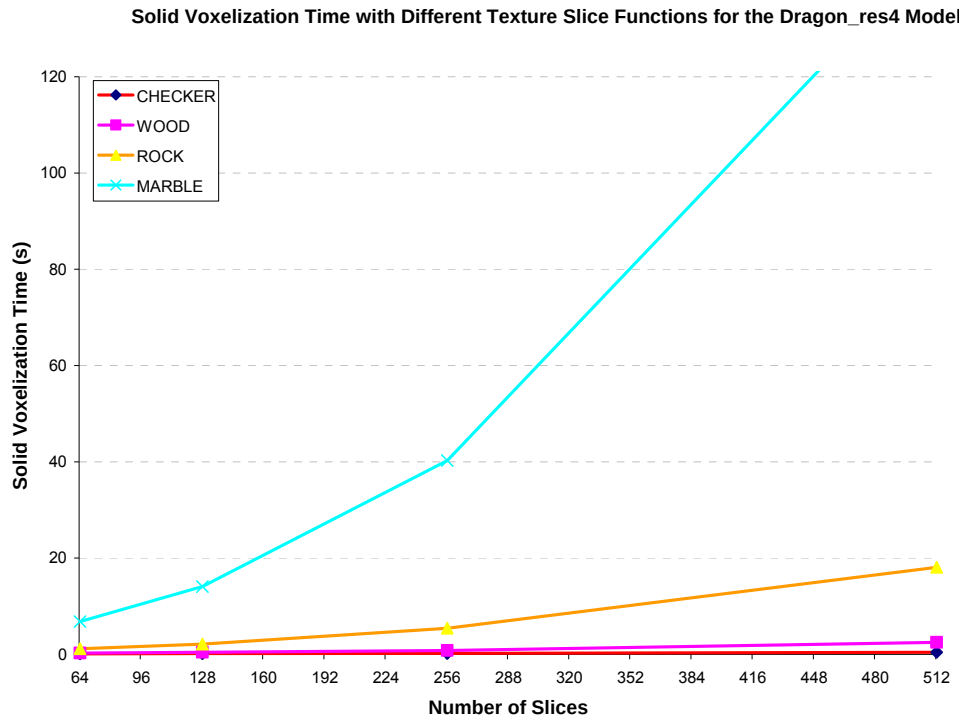


Figure 3.18: Solid voxelization time of the Dragon_res4 model using different texture slice functions

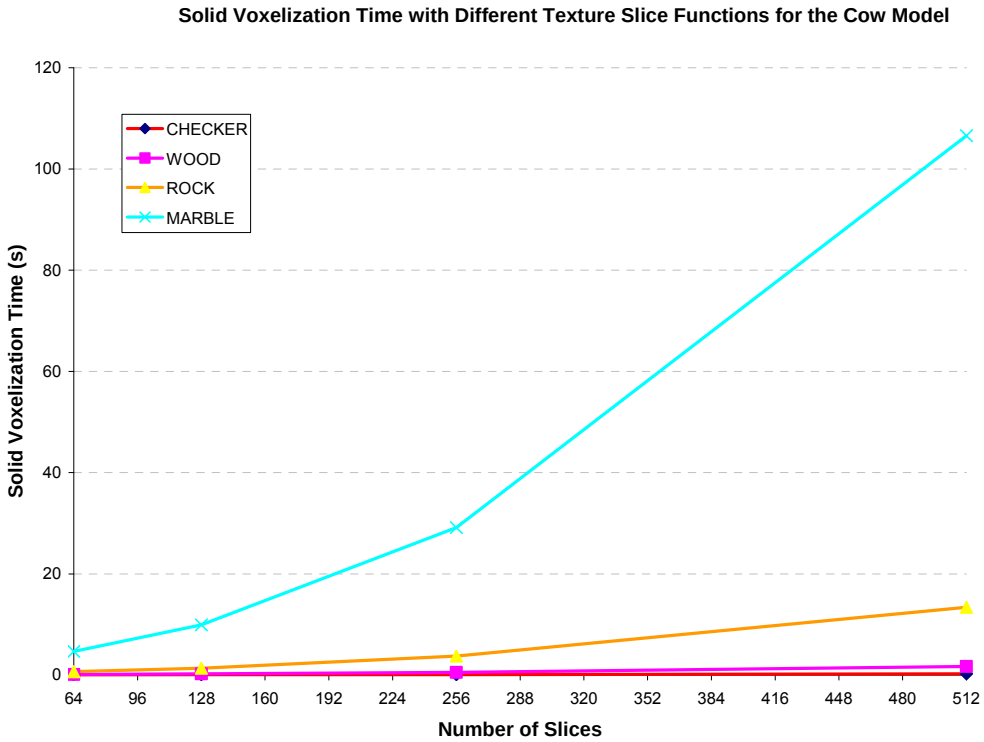
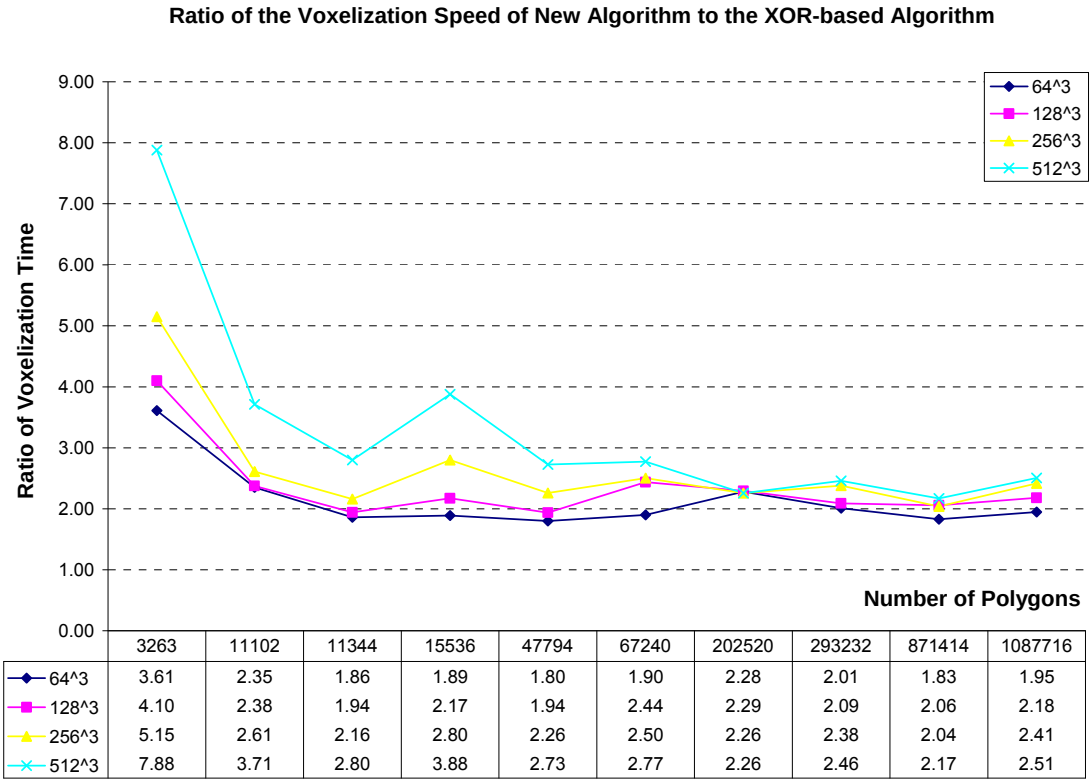


Figure 3.19: Solid voxelization time of the cow model using different texture slice functions



	3263	11102	11344	15536	47794	67240	202520	293232	871414	1087716
64^3	3.61	2.35	1.86	1.89	1.80	1.90	2.28	2.01	1.83	1.95
128^3	4.10	2.38	1.94	2.17	1.94	2.44	2.29	2.09	2.06	2.18
256^3	5.15	2.61	2.16	2.80	2.26	2.50	2.26	2.38	2.04	2.41
512^3	7.88	3.71	2.80	3.88	2.73	2.77	2.26	2.46	2.17	2.51

Figure 3.20: The ratio of the voxelization speed of the new algorithm to the XOR-based algorithm

3.6 Advantages of GPU-based Real-time Solid Voxelization

There are several advantages of the real-time solid voxelization approach as follows:

- fully implements 3D scan conversion (i.e. voxelization) by GPU acceleration
- uses stencil masking and surface parity check techniques for faster and more flexible solid voxelization
- only once efficiently creating the multi-valued volume during the voxelization, which is much different from prior inefficient multi-valued volume computation methods, usually first generating binary volume and then applying the value to each pixel.

- creatively uses different dynamic solid slice functions to generate various kinds of special effects for interior features in the volume during the voxelization. Furthermore, any type of voxels from binary to RGBA with transparency can also be generated.
- significantly improves the speed for solid voxelization in real time
- be able to generate any size of the volume theoretically. In practice, the maximum size of the solid slice is the maximum of the viewport in the frame buffer.
- avoids missing voxels and erroneous voxels
- be suitable for both convex and concave geometric objects
- efficient, robust, flexible, easy to implement, and convenient to integrate into other applications
- independently voxelizes each slice, which is critical for parallelization. Chapter 4 and Chapter 5 will discuss one important parallelization method using this feature to improve the total performance.

4 A New Multi-Core Pipelined Architecture by Forced Interrupts

Multi-core architectures have started a new era of computing and boosted performance and efficiency of parallel programs; however, parallel programming on multi-core processors has become the industry's biggest software challenge. Since multi-core hardware architectures are changed to parallel structures, single-processor based software has to be optimized or even rewritten with much work to meet the hardware constraints. For current parallel programming, the development has cost extra time and effort. Furthermore, most of the existing application base is still composed of sequential applications. How to execute those sequential programs efficiently and reliably on multi-core is still remaining a critical and challenging problem.

If we can change the hardware to eliminate the constraints, however, most of the existing single-processor software programs may be able to be directly used with nearly no change. For this purpose, this chapter proposes a new architecture using multi-core pipelining based on program slicing by forced interrupts, and a distinct memory/cache dynamic management technology for parallel computing. This architecture provides a simple and effective solution to on-the-fly computations by transferring the operating states from processor to processor. The most important advantage is that it requires practically the same software as is currently used on a single-CPU system (i.e. a sequential computer), instead of conventional parallel computing methods, such as threading, load balancing, scheduling, etc.

This chapter is organized as follows: Section 4.1 describes the original multiprocessor pipeline based on program slicing by forced interrupts. Especially, several limitations are summarized and discussed. Accordingly, in the following sections, the new multiprocessor or multi-core pipeline is proposed to significantly overcome all these limitations and make it practical. Specifically, Section 4.2 describes the crossbar switch based multi-core memory and cache architecture. Section 4.3 and Section 4.4 discuss the new multi-core pipeline organization and timing diagram in detail, respectively. Program requirements are described in Section 4.5. Section 4.6 gives the detailed performance analysis, including time complexity, performance, and experimental analysis. Finally, the summary and advantages are addressed in Section 4.7.

4.1 Original Multiprocessor Pipeline by Forced Interrupts

The original multi-processor pipeline presented in [Ber94, Ber00] can process an information flow progressively in a helicoidal pattern by relocating portions of incoming data. In this pipeline organization, processing intervals of equal duration are obtained by slicing the program by forced interrupts. After an interrupt, a processor relocates its data down the pipeline. The program's computations on a partially processed chunk of data are continued at the next processor. Meanwhile, the vacated processor is thus freed to acquire more incoming data. In the pipeline, chunks of data move straight across a row of processors with each processor working on the data for an interval of time. The processors take turns Loading (L), Processing (P), and Unloading (U) the data in a helicoidal pattern as illustrated in Figure 4.1. This pattern ensures that the incoming data flow will not be interrupted.

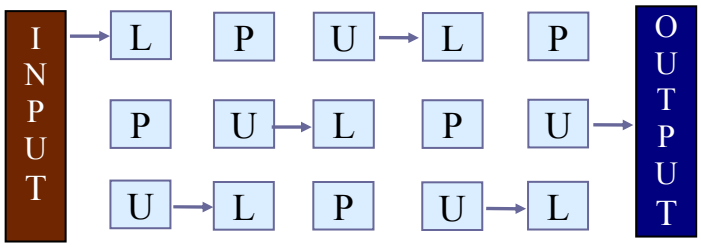


Figure 4.1: Helicoidal pattern in on-the-fly computations by a microprocessor pipeline (Loading, Processing, and Unloading rotate in the columns in the cycle: L→P→U→L)

The multiprocessor pipeline allows an arbitrary algorithm to be performed on-the-fly on a data chunk, given a sufficient number of processors. This means that if an algorithm can be performed by a conventional microprocessor under static conditions, it can be performed on the multiprocessor pipeline.

However, there are several limitations of the original multiprocessor pipeline as follows:

- Data relocation

  Frequent data relocation especially for large data block applications like 3D graphics can cause big overhead costs leading to the overall performance decrease.

- Data overlapping

  The system described in [Ber94] uses data overlapping to solve the problem of processed data chunks across the segmentation imposed by the buffer size. The amounts of memories of the processors have to be occupied by these duplicate (i.e. overlapped) data.

- One relocation allowed at a particular time

To reduce the bus traffic, the data relocations, i.e. the loadings/unloadings, are arranged so that only one data chunk relocates in one shared bus at a particular time.

- Special handling for the data required longer pipeline to process

  There are two ways proposed to handle this case in [Ber94]. One way is to provide enough pipeline length for even the longest processing time. This is not practical. The other way is to take the intermediate results from the end of the pipeline and either sending them to some overflow processing facility or accumulating them and sending them back when the incoming data stream ceases.

In this dissertation, a novel multiprocessor (i.e. multi-core) pipelined architecture by forced interrupts is proposed with a distinct memory/cache management technology to significantly overcome all these limitations. It is also successfully applied to parallel solid voxelization as discussed in Chapter 5. Note that the core generic forced-interrupt driven multiprocessor pipelining technique in the original pipeline is still employed in the new architecture system.

4.2 New Multi-core Memory and Cache Architecture Based on Crossbar Switch

The memory and cache management is very important for multi-core systems. A novel memory/cache management technology described in this section is one of core parts of this new architecture design. It significantly improves total performance in both space and time. Therefore, this part is introduced first.

Since data blocks could be very large in the parallel applications, if DMA (Direct Memory Access) is used to move big data blocks from the memory in the previous processor to the next processor frequently at each time, the accumulated overhead costs cannot be ignored. To reduce the data relocation costs and bus limitations, the new architecture use a crossbar switch based dynamical management technique for both memories and caches to avoid relocating the data down the pipeline from one processor to another processor. The new memory and cache architecture for the technique is illustrated in Figure 4.2.

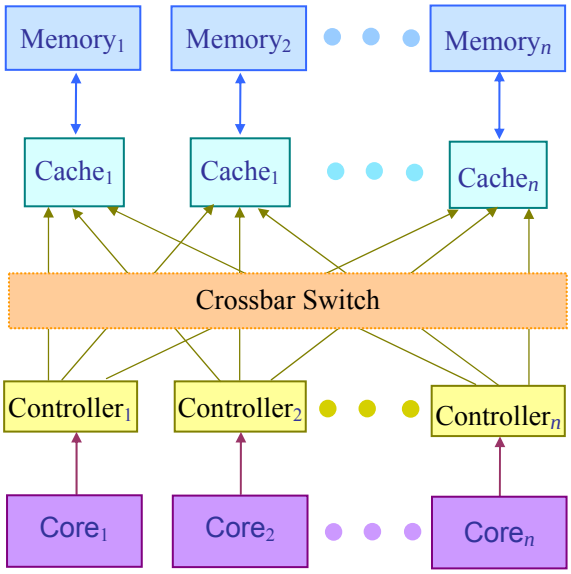


Figure 4.2: Crossbar switch based Multi-core memory and cache architecture

Each processor/core does not have a fixed memory and cache as does an ordinary processor. Instead, each of them will be assigned to connect to a given memory and cache at a given time. There is only one bus between such memory and cache. Both of them can be regarded as a group. The number of the groups is the same as the number of cores.

Each core has its own controller, which is employed to switch the entire memory and cache, i.e. one group, for the previous processor into the next processor. Once the group of previous processors is switched to the next processor, the current group of the previous processors needs to be cleared for use.

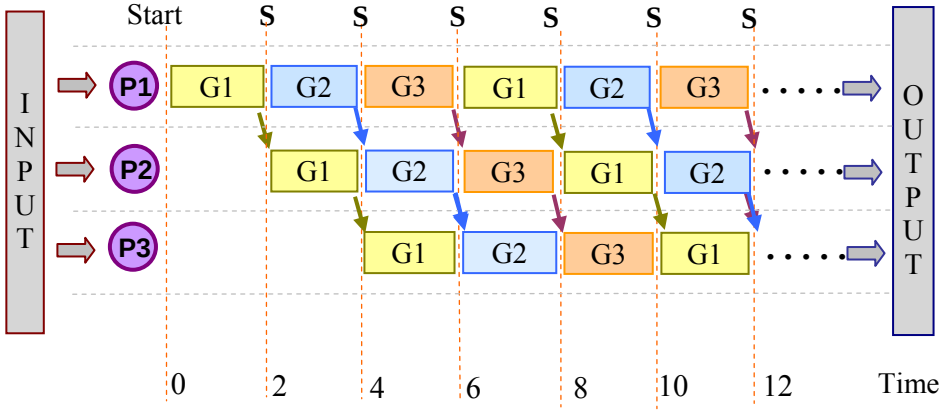
A crossbar switch is the key to carrying out this dynamic memory/cache management technology. It exchanges two memory and cache groups between two processors at a given time. Correspondingly, the bus crossbar is used for connecting all the cores to all the groups to guarantee data transmission between them at full speed and with no contention. Although the crossbar used for this architecture is similar to Nvidia's LMA, they are different in principle. The purpose of the crossbar in our architecture is mainly for memory and cache switch apart from high-bandwidth data transmission. Furthermore, from the view of cores, the "N→N" connection relationship is for all cores and groups. This architecture differs from Nvidia's LMA, in which one core has multiple memory controllers connecting to their corresponding memory banks. Actually, from the view of cores, the "1→N" connection relationship relates a given core to all its memory banks.

Additionally, the bus crossbar used in the new architecture can also reduce large bus traffic because each memory and cache group has a dedicated connection to one processor. This not only reduces the bus traffic, but also eliminates the constraint that the data movement has to be arranged at a particular time as mentioned before.

4.3 New Multi-core Pipeline Organization

The new multi-core pipeline architecture does not relocate the data down the pipeline as would be the case for the original multi-core pipeline when an interrupt is forced.

Instead, it only switches the memory and cache group of the previous processor, where the data have been loaded and/or processed, into the group of the next processor. These data also contain the small amount of the data of the operating state, such as Program Status Word (PSW) and all registers with the program counter, so that the next processor could resume operations where the previous processor stopped.



G = Group (memory and cache)
S = Switching (on/off for the connection between each processor and each group)
P1 = Processor 1 P2 = Processor 2 P3 = Processor 3

Figure 4.3 Improved multi-core pipeline organization and data flows

The new multi-processor pipeline organization and data flow are depicted in Figure 4.3. The major improvement is to replace Unloading (U) with Switching (S) in the organization. However, Loading (L), Processing (P), and Switching (S) do NOT rotate in the columns in the cycle: L→P→S→L in a helicoidal pattern as does the above original pipeline. Instead, the Switching (S) for all the processors occurs at the same time in the new multi-processor pipeline. Accordingly, Loading (L), Processing (P), and Switching (S) may rotate in the columns in the cycle: L→P→**S**, P→P→**S**, or P→L→**S** as shown in Figure 4.4.

In Figure 4.3, the vertical orange dashed lines show the cycles. Each small rectangle represents one operation in one cycle. All blocks and arrows with the same colors describe the flows of the data chunks. P1, P2, and P3 describe the processors or cores. In theory, there could be any number of processors, depending on the applications and user requirements.

Because the architecture does not specify a particular processor to perform a particular operation, data-dependent branching of the algorithm does not require special handling. A processor working on a particular data chunk behaves just as a standard processor, resulting in variable-length processing times for the data chunks. If a chunk becomes fully processed before the end of the pipeline, the result can be withdrawn. Data which require longer processing can be switched back to connect the first processor to continue processing. This is totally different from the original multiprocessor pipeline [Ber04], which need to send these data to some sort of overflow processing facility, or accumulate them and send them back through the pipeline when the incoming data stream ceases. This new method would be advantageous for highly variable data processing times. The next section also gives more details to explain this using an example in Figure 4.4.

4.4 Timing Diagram

Figure 4.4 shows the timing diagram of an example of a 3-core pipeline. It is assumed that Switching (S) takes one cycle. In fact, S may take more than one cycle or less than one cycle, which depend on hardware and software. Because of the dynamic memory/cache management with the crossbar-based techniques, all the Switchings (S)

occur at the same time. That is, all the processed data from previous processors to next processors are switched at the same time.

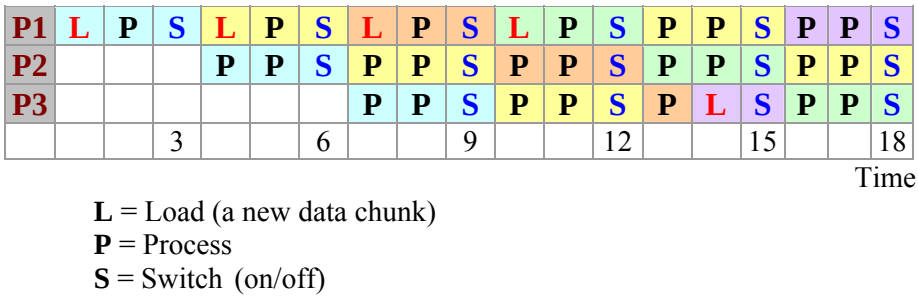


Figure 4.4: Timing diagram of a 3-core pipeline system

In the figure, each color block represents a data chunk being processed in different processors in different time. The same color blocks indicate the data flow for one chunk. A new data chunk can be loaded (L) by any one of the processors as long as the previous data chunk is finished processing. In other word, any one of the processors can load the data as long as it is free. The various lengths or processing time of the data chunks can thus be automatically scheduled onto the processors without considering any load balancing or scheduling issues.

In fact, one data chunk is always being processed within this memory and cache group although the core connected to this group is changed at every switching time. That is to say, the data do not move while the core or processor moves. This is the major difference from the original multiprocessor pipeline, in which the data chunk has to be relocated when an interrupt is forced.

The new architecture is much more efficient in both space and time. The overall performance can be improved significantly. This is totally different from the

multiprocessor pipeline system in [Ber94, Ber00], which allows only one data chunk relocation at a particular time on one shared bus.

4.5 Program Requirements

An important feature of this architecture is that it uses practically the same software as a sequential computer. A program for this system can be developed on an ordinary sequential computer. Each processor is distributed with the same application program. To run on the system, the program would just have to incorporate some interrupts. The interrupt processing can be done by the operating system, compiling systems, hardware triggers, or even defined by users.

A processor working on a particular piece of data, upon an interrupt, will switch its memory to the next processor. The processor will also pass the PSW so that the next processor could resume operations where the previous processor stopped. The formation of the PSW in the system is similar to the routine procedure of formatting the PSW for interrupts in ordinary microprocessors.

4.6 Performance Analysis

4.6.1 Time Complexity

Clearly, the time complexity of the algorithm based on the new multi-core pipelined architecture will only affect the total length of the pipeline including the overhead of the memory and cache group switches between processors after forced interrupts.

Let me analyze how this new multi-core pipelined architecture improves the overall performance for parallel computing. The principle of the *N*-core pipeline is descried in Figure 4.5.

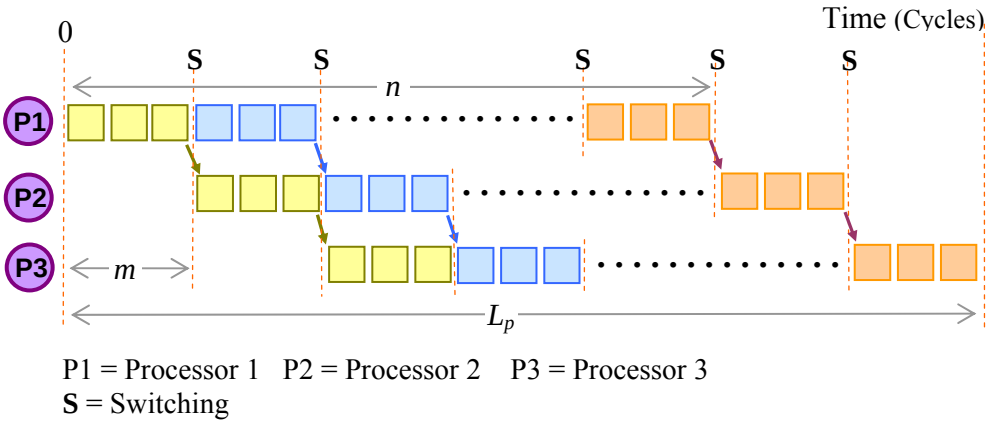


Figure 4.5: The principle of the *N*-core pipeline

For the multi-core pipeline system as shown in Figure 4.5, to better explain the performance, the pattern of the system operations is assumed to be discretized into cycles with each cycle, C. In fact, it can be discretized into much smaller time unit in modern architectures so that the architectures can be programmable.

Let us consider a program to process a dataset containing data chunks with equal or various sizes. Processing the entire dataset requires K cycles. On a conventional processor, the execution time of a program, T_c, can be expressed by

$$T_c = KC \tag{4.1}$$

In Figure 4.5, let N be the number of processors/cores. If the entire dataset can be processed on all N cores, on the average, the number of cycles on each processor/core, n, can be given by

$$n = \frac{K}{N} \tag{4.2}$$

Thus, the total time spent on the conventional processor T_c can also be described in combination with above two equations as

$$T_c = nNC \tag{4.3}$$

For the N-core pipeline, let *m* be the number of cycles for each interval of the equal duration in the pipeline. Thus, the length of the *N*-core pipeline for processing of data, L_p, can be described by

$$L_p = n + (N-1)m \tag{4.4}$$

Let *q* be the number of the internals for the entire pipeline, it can be

$$\begin{aligned} q &= \frac{L_p}{m} \\ &= \left\lceil \frac{n+(N-1)m}{m} \right\rceil \\ &= \left\lceil \frac{n}{m} \right\rceil + (N-1) \end{aligned} \tag{4.5}$$

Since switching by crossbar may take some extra time, let *d* be the number of cycles of processing and delays or latencies for each switching based on hardware. So the total number of cycles of the overhead of the context switching in the *N*-core pipeline, can be expressed as

$$\begin{aligned} H &= (q-1)d \\ &= (\frac{L_p}{m} - 1)d \\ &= (\left\lceil \frac{n}{m} \right\rceil + N - 2)d \end{aligned} \tag{4.6}$$

Thus, in the *N*-core pipeline system, the total length of the pipeline, noted by *L*, to process the same size of the data including the overhead of content switching, can be described by

$$
\begin{aligned}
L &= L_p + H \\
&= qm + (q-1)d \\
&= n + (N-1)m + (\left\lceil \frac{n}{m} \right\rceil + N - 2)d
\end{aligned}
\tag{4.7}
$$

The total time spent T on this multi-core pipelined system can be obtained by

$$
T = LC \tag{4.8}
$$

4.6.2 Performance

Let me first calculate the speed-up of the execution time on the new N-core pipeline system compared to the time on one conventional processor.

Combined with Equation (4.1), Equation (4.3), Equation (4.4), Equation (4.7), and Equation (4.8), the speed-up is given by

$$
\begin{aligned}
Speedup &= \frac{T_c}{T} \\
&= \frac{K \cdot C}{(qm + (q-1)d) \cdot C} \\
&= \frac{K}{qm + (q-1)d} \\
&= \frac{nN}{n + (N-1)m + (q-1)d} \\
&= \frac{nN}{n + (N-1)m + (\left\lceil \frac{n}{m} \right\rceil + N - 2)d}
\end{aligned}
\tag{4.9}
$$

Since n is often much larger than N and m, so the formula in Equation (4.9) can be simplified as

$$Speedup = \frac{N}{1 + \frac{(N-1)m}{n} + \frac{(\left\lceil \frac{n}{m} \right\rceil + N - 2)d}{n}} \approx \frac{N}{1+d} \tag{4.10}$$

Obviously, when the number of cores N increases, the speed-up or performance increases correspondingly. That is, the execution time on the N-cores pipelined system decreases as N increases. Since the *Speedup* must be greater than 1 to guarantee the good performance, so

$$Speedup \approx \frac{N}{1+d} > 1 \Rightarrow N > d+1 \tag{4.11}$$

Thus, N must be greater than *(d+1),* the performance can be guaranteed to be $Speedup > 1$. That is, the minimum number of cores N should be greater than (*d+1)* to guarantee $Speedup > 1$. In some conditions, $N < d+1$ may reach $Speedup > 1$.

4.6.3 Experimental Analysis

I developed a tool to evaluate this new architecture. In this section, some experimental results are reported based on this architecture with various overheads taken into account.

In the first four experiments as shown in Figure 4.6, Figure 4.7, Figure 4.8, and Figure 4.9, we assume the total execution of a given sequential program takes 10,000 cycles and one-time switching overhead *d* takes 2 cycles.

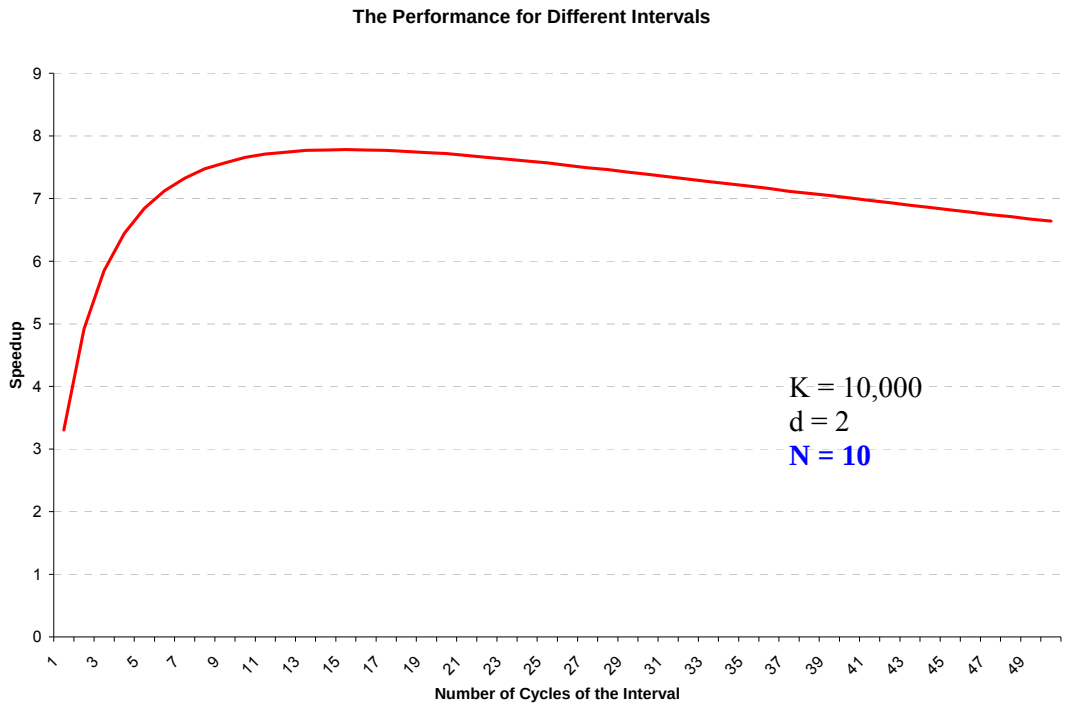


Figure 4.6 The performance for different Intervals

Figure 4.6 shows the performance increases with interval time *m* increasing for fixed number of cores, *N*, and switching time, *d*, and reaches the maximum then decreases slowly. This is because the pipeline will become longer if *m* becomes bigger. The performance is affected by the length of the pipeline.

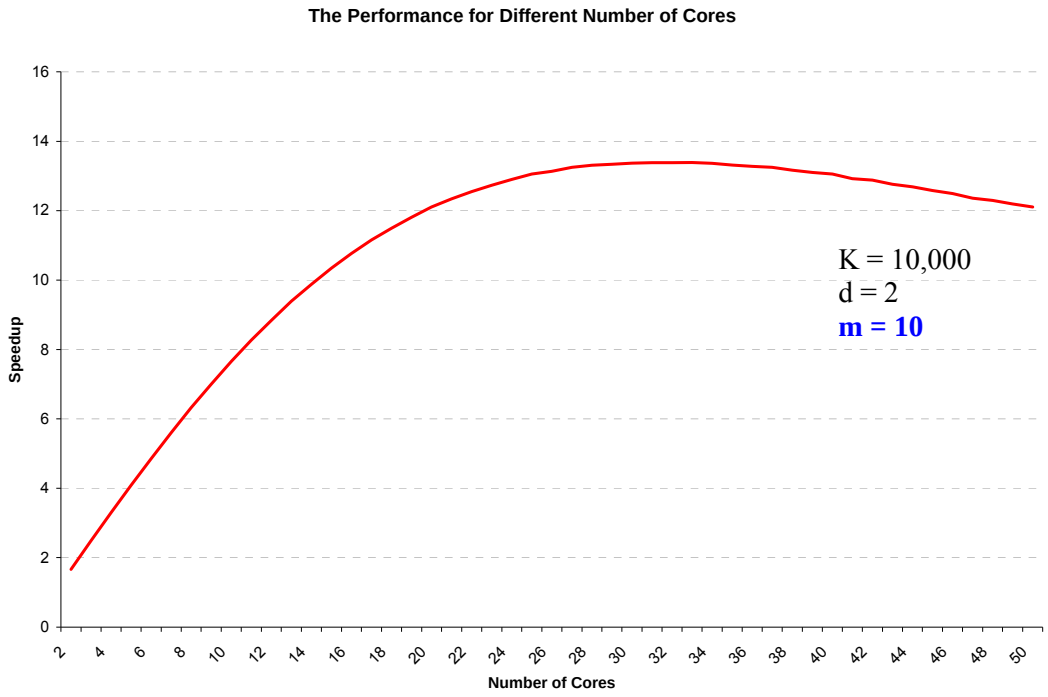


Figure 4.7: The performance for different number of cores

Table 4.6.1: The performance for different number of cores with corresponding pipeline lengths

N	*n*	*L*	*SpeedUp*	*N*	*n*	*L*	*SpeedUp*	*N*	*n*	*L*	*SpeedUp*
2	5000	6010	1.66	**19**	526	846	11.81	**36**	277	751	13.28
3	3333	4023	2.49	**20**	500	826	12.11	**37**	270	754	13.25
4	2500	3034	3.30	**21**	476	810	12.34	**38**	263	759	13.17
5	2000	2446	4.09	**22**	454	796	12.55	**39**	256	762	13.10
6	1666	2058	4.86	**23**	434	784	12.73	**40**	250	766	13.06
7	1428	1784	5.60	**24**	416	774	12.90	**41**	243	771	12.92
8	1250	1582	6.32	**25**	400	766	13.06	**42**	238	776	12.88
9	1111	1429	7.00	**26**	384	760	13.14	**43**	232	782	12.76
10	1000	1306	7.66	**27**	370	754	13.25	**44**	227	787	12.69
11	909	1209	8.27	**28**	357	751	13.31	**45**	222	794	12.58
12	833	1131	8.84	**29**	344	748	13.34	**46**	217	799	12.49
13	769	1065	9.39	**30**	333	747	13.37	**47**	212	806	12.36
14	714	1012	9.88	**31**	322	746	13.38	**48**	208	812	12.30
15	666	966	10.34	**32**	312	746	13.38	**49**	204	820	12.19
16	625	929	10.76	**33**	303	**747**	**13.39**	**50**	200	826	12.11
17	588	896	11.16	**34**	294	748	13.36				
18	555	869	11.50	**35**	285	749	13.32				

Figure 4.7 shows the performance increases with number of cores increasing for fixed interval time, *m,* and switching time, *d*, and reaches the maximum, then decreases slowly. This is also because the pipeline may increase slowly while the number of cores becomes big as shown in Table 4.6.1. In the table, we can see the minimum length of the pipeline is 747 when the number of cores is 33. At this time, the performance speedup reaches the maximum value, 13.39.

That is to say, for a given program with fixed interval time, *m,* and switching time, *d*, the performance may not reach the maximum for some numbers of cores. Put another way, if given an appropriate interval time *m* and switching time, *d*, the performance can reach the maximum speedup. In practice, since *d* is usually fixed for such a multi-core system due to the crossbar switch, the maximum *speedup* can be obtained with an appropriate *m*.

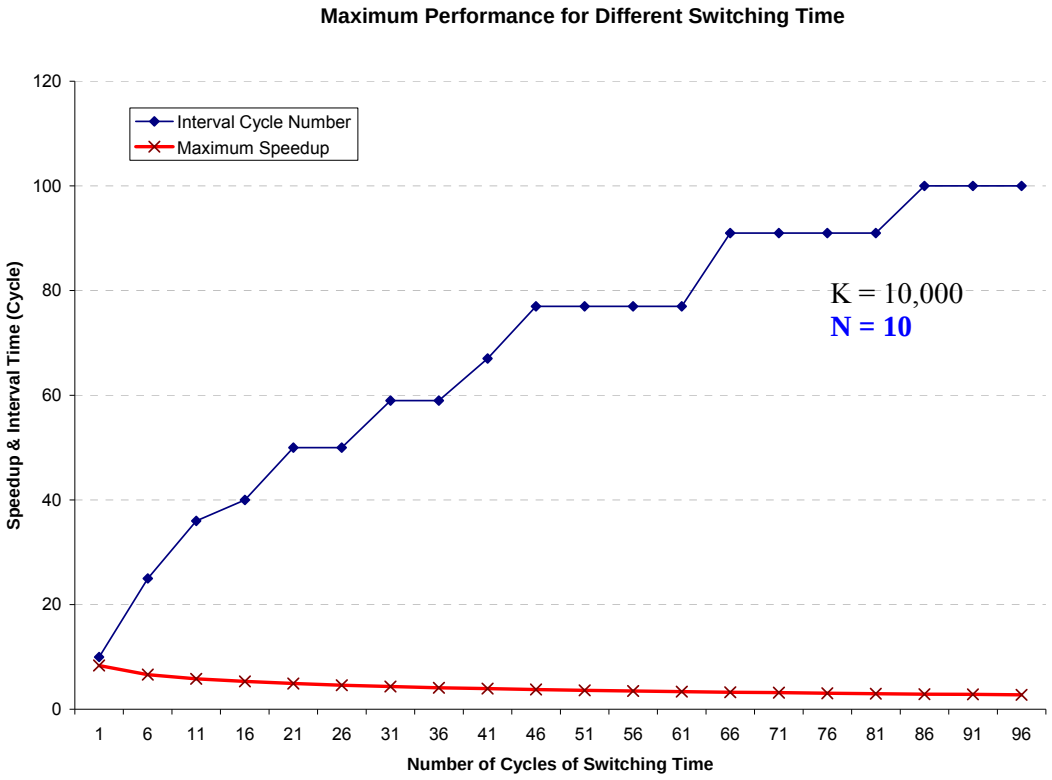


Figure 4.8: Maximum performance for different switching time

Figure 4.8 shows the maximum performance for different switching time, *d*. In the figure, the blue line shows the interval time, *m*, and the red line shows the maximum performance, *max_Speedup*. For a given program and fixed number of cores, *N*, with the increase of the switching time, *d*, the best interval time, m_b, to reach the maximum performance, *max_Speedup*, increases when the switching time, *d*, increases. However, the *max_Speedup* decreases slowly accordingly. This clearly indicates the pipeline length increases with the increase of both interval time and overhead of content switching (i.e. *m* and *d*). Accordingly, the performance decreases.

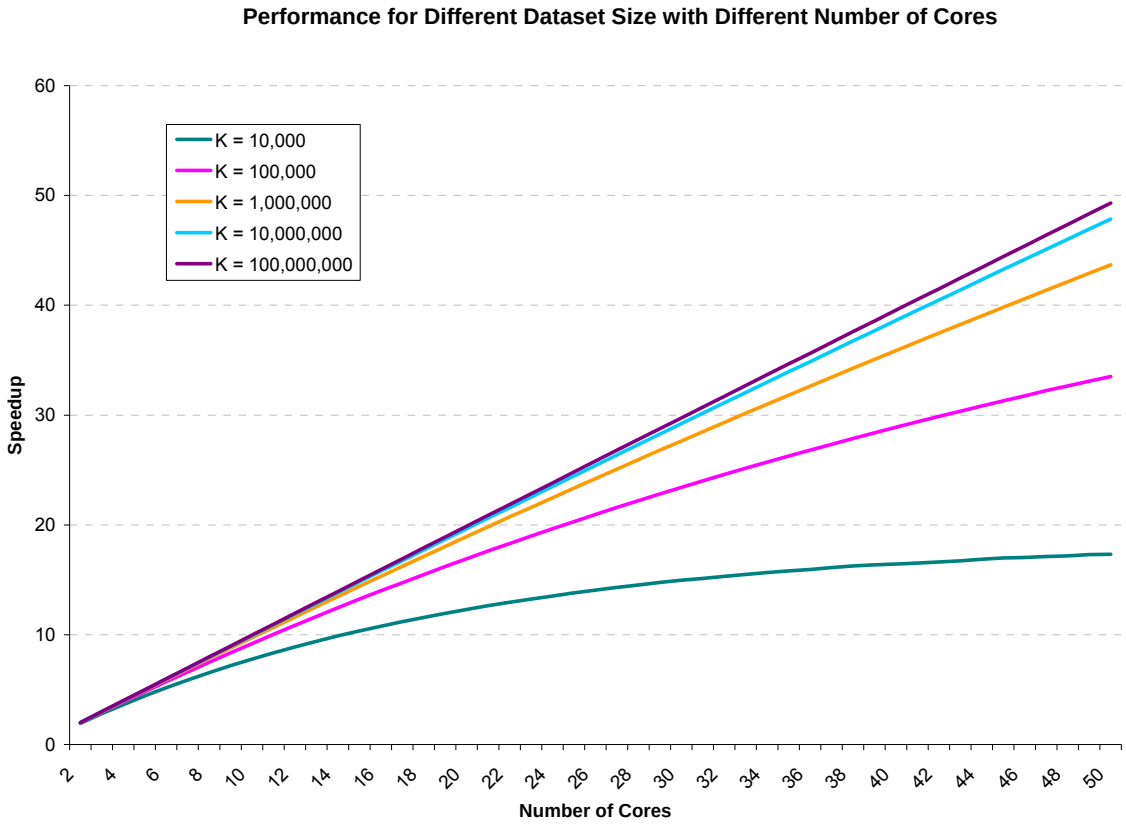


Figure 4.9: The performance for different dataset size with different number of cores

Figure 4.9 shows the performance *Speedup* increases with increasing the number of cores and the size of the program. Note that the *Speedup* is the maximum, *max_Speedup,* in this experiment. The figure clearly indicates when the application program becomes larger, the *Speedup* increases linearly with the number of cores. That is, the *Speedup* is almost close to the number of cores for the large-size applications. This indicates the performance increases the times of number of cores, which is the maximum performance for any multi-core system in theory.

Figure 4.10 shows the best interval time, m_b, to reach the maximum performance decreases and then become stable with the increase of the number of cores. Furthermore, all the best interval time, m_b, for different size of the application programs is close to each other when the number of cores increases. This implies that the most appropriate interval time, m, can be chosen to maximize the performance for all the application programs on a multi-core system with the certain amount of cores. This is a trade-off for a multi-core system to ensure the best performance for all the applications.

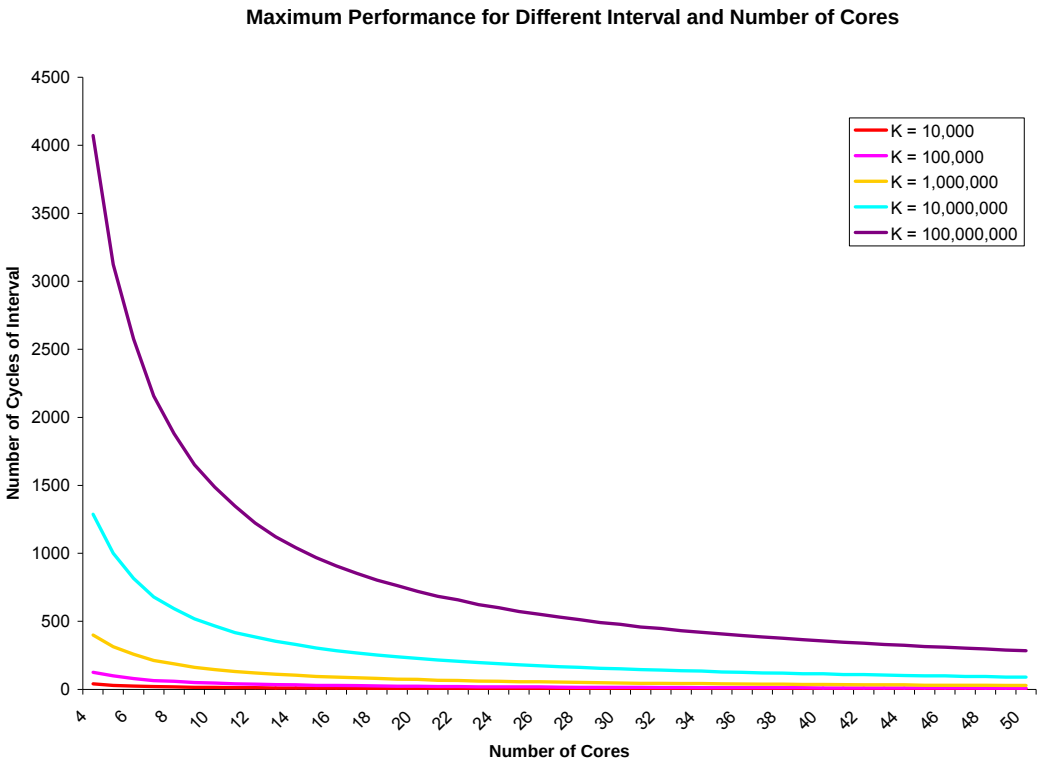


Figure 4.10: Maximum performance for different intervals and number of cores

In summary, with the increase of the number of cores, the performance increases totally. However, choosing the best interval time, m_b, is the key to make full use of all the

cores to maximize the performance for all the application programs. Such a possible m_b can be found for all the application programs based on the analysis of Figure 4.10. In addition to selecting one fixed best interval time, m_b, for the entire system, the dynamical best interval time, m_b, can also be automatically assigned to each application program according to the total application and data size, the number of cores, and the switching time on a multi-core system while the program is compiled.

4.7 Summary and Advantages

This dissertation presents a new multi-core pipelined architecture based on core forced-interrupt driven multiprocessor pipelining and switch-based dynamic memory and cache management techniques. The proposed architecture provides a simple and effective implementation for on-the-fly parallel computing by switching the entire data from core to core. It does not need conventional complicated parallel computing methods, such as load balancing, scheduling, etc. This is exactly the advantage of this proposed architecture.

Obviously, the new multi-core pipeline architecture significantly overcomes all these limitations of the original multiprocessor pipeline as described in Section 4.1. More specifically, the new pipeline advantages over the original pipeline limitations are summarized as follows:

- No data relocation

  Content switching is employed to minimize big overhead costs due to data relocation while an interrupt is forced.

- No data overlapping

Due to content switching, no processed data chunks across the segmentation.

- All the Switches (S) are forced by interrupts at a regular time

All the data are switched to their corresponding processors. However, the original multiprocessor pipeline only allows one data relocation at a particular time due to one shared bus.

- No special handling for the data required longer pipeline to process

The new architecture does not need special handing for the data required longer pipeline to process. But, the original multiprocessor pipeline need overflow facility or accumulation and sending back while the data stream ceases.

Additionally, more specific advantages for the new switch-based dynamic memory and cache-management technology in the new architecture are emphasized as follows:

- avoids moving the data from one memory to another memory
- allows more than one Switching operation at a time; all the Switchings occur at the same time
- reduces bus limitations and large bus traffic using the bus crossbar
- improves performance in both space and time

Finally, to summarize, there are several overall advantages of this new multi-core pipelined architecture as follows:

- provides the continuous data processing of intensive information flows
- requires essentially the same software as ordinary sequential algorithm
- avoids load balancing and scheduling
- avoids the need for synchronization among the processors
- avoids busy waiting of processors on a spin-lock

- avoids the duplication for incoming data stream
- suitable for highly variable data processing time

Another advantage worth mentioning is that this core multiprocessor pipelining technology also provides an important solution to processing the continuous intensive information flows without limitation by size. Consequently, this would be very helpful to real-time massive data processing.

5 Parallel Solid Voxelization Using Multi-Core Pipelining

A parallel solid voxelization system using multi-core pipelining presented in this chapter is based on the new multi-core pipelined parallel architecture described in Chapter 4. This system is designed to slice the program by forced interrupts onto the multiple cores or processors for processing the continuous flow of intensive computations. Through this pipeline architecture, the new real-time solid voxelization program for a conventional single-CPU computer with 3D accelerator (GPU) described in Chapter 3 will be automatically scheduled onto multiple processors with forced interrupt control. Consequently, the speed, quality, and size of the voxelization can be improved significantly.

In this chapter, Section 5.1 extends the generic multi-core pipelined parallel architecture presented in Chapter 4 to the multi-core GPU architecture for parallel solid voxelization. The comparison with current multi-core GPUs is also discussed. The flow chart of parallel solid voxelization is also described. Section 5.2 presents a parallel solid voxelization system based on the new multi-core pipelined GPU architecture. The system architecture, parallel voxelization algorithm and comparison with conventional parallel voxelization are discussed in detail. Section 5.3 describes the principle of real-time parallel solid voxelization by multi-core pipeline in detail. Section 5.4 discusses the

performance and experimental analysis and comparison of the multi-core pipelined parallel voxelization system. The summary and advantages are concluded in the end.

5.1 A Multi-core Pipelined GPU Architecture

Based on a generic multi-core pipelined parallel architecture as described in Chapter 4, a multi-core pipelined GPU architecture is extended for parallel graphics and visualization, especially for parallel solid voxelization in this chapter.

5.1.1 The Principle of the New GPU Architecture

In the multi-ore pipelined GPU architecture as shown in Figure 5.1, each generic memory is replaced with a graphics memory in each group and each generic core is replaced with a standard GPU core from the generic multi-core pipelined parallel architecture as shown in Figure 4.2.

Note that the graphics memory contains stencil buffer, depth buffer, color buffer, and other graphics buffers as the same as those in a standard GPU system. In other word, these buffers are local to a certain processor at a certain time. From the functional point of view, one core, graphics memory, and cache compose of a temporary standard GPU system when one memory and cache group is switched to a certain core at a certain time. One certain data chunk is always being processed within this memory and cache group although the GPU core is changed at every switching time. That is, the data do not move while the core or processor moves.

Additionally, the sequential GPU program can be directly applied to this parallel GPU architecture without any change. This is the biggest benefit over the existing multi-

core GPU systems, in which the development of current widely used GPU parallel programming in CUDA has shown extra time and effort costs.

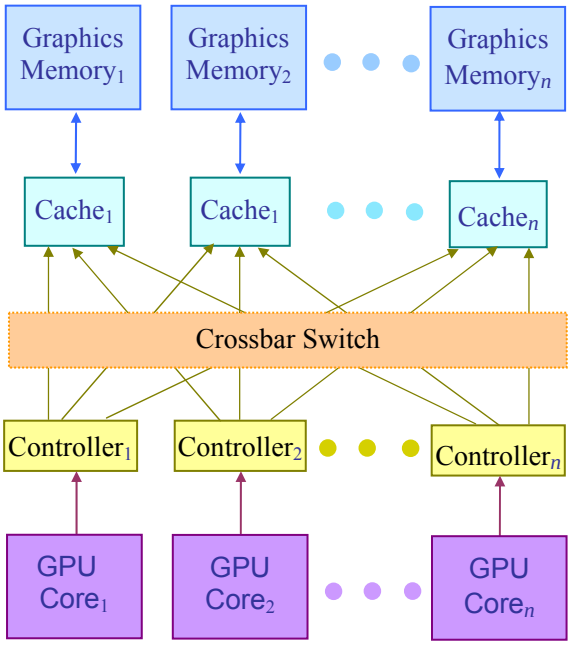


Figure 5.1: Crossbar switch based Multi-core GPU architecture

5.1.2 Comparison with Existing Multi-core GPUs

As compared with current existing multi-core GPUs, there are several different features in the new GPU architecture. For example, each processor in the new GPU architecture can execute MIMD mode because of multi-core pipelining. But, in current GPUs, the SIMD mode is executed. Moreover, the multiprocessors themselves execute asynchronously, and without communication. The new architecture, however, guarantees communication between GPUs through memory and cache switching, just as is the case for data processing within one processor.

Table 5.1.1: Comparison of multi-core GPU architectures

Category	Existing Multi-core GPUs	Mulit-core Pipeline GPUs
Parallel structure	Data parallelism	Pipeline parallelism by program slicing with forced interrupts
Programming model	Multithreaded	No threading required
Computing mode	SIMD	MIMD
Synchronism	Multiprocessors execute asynchronously	Multiprocessors execute synchronously to switch the memories including operating states by forced interrupts
Communication between multiprocessors	No.	Yes. Communication through memory and cache switches.
Program requirements and data management	Decompose the program into kernels and manage the data in corresponding domains	Practically same as ordinary sequential program
Data dependency	Independent	Independent or/and dependent
Data requirements	Structured data	Any type of data
Crossbar connection	"1→N": used within one core from one core to all its own memory banks	"N→N": used among all the cores from all cores to all their entire memories and caches.

Another important difference between them is the parallel structure as well as the programming model. The current GPU is designed based on data parallelism. This requires the developers to split the computational kernels (i.e., functions) and corresponding domains. Data movement needs to be managed between the memories and the host. Each kernel will complete its entire domain before the next kernel starts. The

parallelism comes from the domain of a single kernel (data parallelism), not from running many different kernels at once (task parallelism). The new GPU architecture provides a method of parallel computing that is as simple and effective as sequential programming without these extra efforts.

Furthermore, the major comparison between the new multi-core pipeline GPU architecture and current multi-core GPU architectures is summarized in Table 5.1.1.

5.2 A Multi-Core Pipelined System for Parallel Solid Voxelization

The multi-core pipelined architecture offers a feasible and effective solution to the problem of real-time solid voxelization by switching the portions of incoming data from core to core in among multiple cores on the fly.

5.2.1 The System Architecture

Such a system for the voxelization algorithm as shown in Figure 5.2 can consist of a plurality of GPU cores, a crossbar switch, and the memory and cache groups with the same number of the cores. The memories are used for processing and storing input geometric object data. Similarly, the voxelization may produce intermediate data. The memories can be used for storing the volumetric data only if the data will be immediately used for other applications, such as volume rendering after voxelization. The caches are used for storing the display lists, groups of frequently used commands that have been stored for later repeated execution in multiple times. Using display lists can reduce the cost of repeatedly re-evaluating and re-transmitting data over and over again to draw each frame. The performance can be much improved by using display lists. Additionally, the

memory and cache group also contains framebuffers, including stencil buffers, depth buffers, etc. That is, each group looks like that in the single-GPU system. The output unit at the end of the pipeline can be hard disks or external frame buffers.

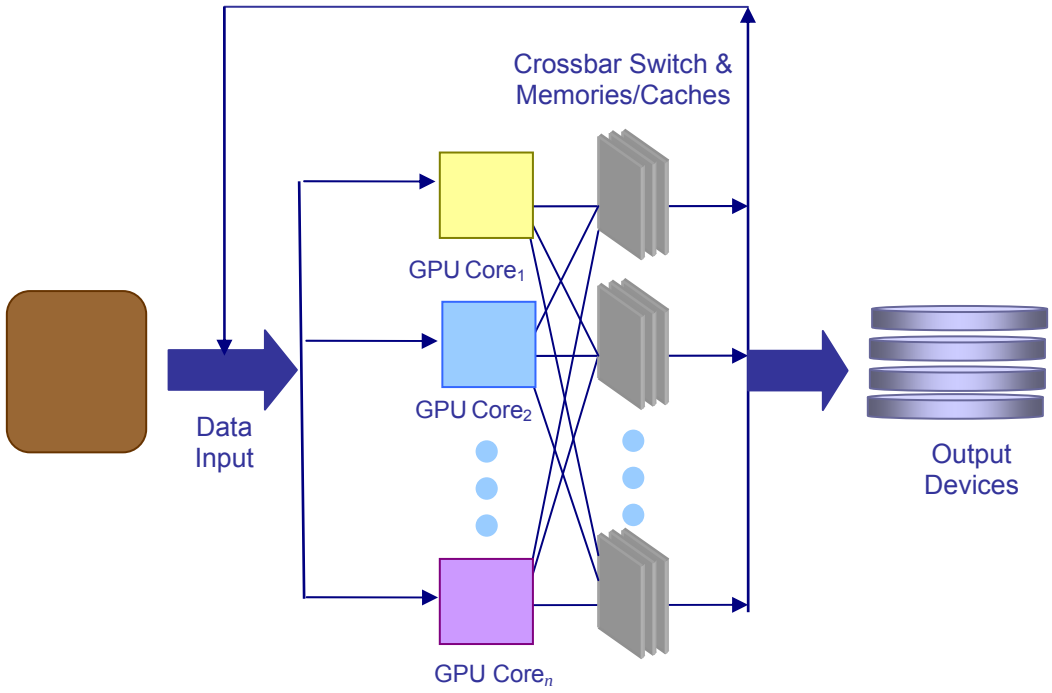


Figure 5.2: Multi-core pipeline system for parallel solid voxelization

All of the required processing of the incoming data can be conveniently performed on-the-fly using the new multi-core pipelined GPU architecture. The new architecture is able to implement on-the-fly the solid voxelization algorithm within a block of data of a given size. From the functional point of view, this pipeline can be treated as a sequence, which contains multiple GPU cores with their dynamic banks of memory, including frame buffers for GPUs.

An important feature of this pipeline is that this system can use practically the same software as a sequential computer. A program for the multi-core pipeline for voxelization can be developed on an ordinary sequential computer.

5.2.2 The Parallel Solid Voxelization Algorithm and Comparison

Because of the benefit of the new multi-core pipelined GPU architecture, the core slice-independent real-time solid voxelization algorithm by GPU acceleration on a sequential computer described in Chapter 3 can be directly used without any change for the parallel solid voxelization using multi-core pipelining. The forced interrupt control can be done by the compiling system or hardware.

In order to better compare with the conventional parallel solid voxelization algorithm, the sequential solid voxelization for the multi-core pipelined parallel system is summarized in Figure 5.3 based on the sequential algorithms for real-time solid voxelization is described in Figure 3.5 and Figure 3.6 in Chapter 3. In this algorithm, stencil buffers and depth buffers will be employed as well. Additionally, an entire volume is computed based on each slice. Each slice is processed independently. Since there is no communication between any slices, there is no need for voxelization of adjacent slices to adjacent processors. This makes the multi-core pipeline simple.

In a word, there is no need to change the sequential code for parallel solid voxelization in the multi-core pipelined system.

Algorithm 5.1: Sequential Solid Voxelization for the Multi-core Pipelined Parallel System

SolidVoxelization_Sequential()

Input:

O ← a closed surface-based object

Nx, Ny, Nz ← size of the output volume

Output:

V← the voxelized solid volume

Begin

1 Set viewport, projection, and background color

2 Generate the display list *L* for geometric object *O* drawing.

3 For each slice *i = 0* to *Nz – 1* **do**

*i*th solidSlice ← CreateSolidSlice (*L*, *i*)

Insert *i*th solidSlice into the 3D texture volume *V*

End

End

4 Output the entire volume *V*

Figure 5.3: Sequential solid voxelization algorithm for the Multi-core Pipelined Parallel System

However, for the conventional parallel solid voxelization, the program has to be changed. There are many different algorithms for convention parallel voxelization. For example, a simple yet inexpensive way of conventional parallelization is to partition the volume into slices and assign them to processors. To more clearly compare, the same core solid slice creation algorithm as described in Figure 3.5 is still used for independently creating each slice in the conventional parallel solid voxelization system.

Figure 5.4 and Figure 5.5 show the conventional parallel solid voxelization algorithm based on slice independence and the flow chart, respectively.

Algorithm 5.2: Conventional Parallel Solid Voxelization

SolidVoxelization_Parallel()

Input:

O ← a closed surface-based object

Nx, *Ny*, *Nz* ← size of the output volume

Output:

V← the voxelized solid volume

Begin

1 Set viewport, projection, and background color

2 Generate the display list *L* the geometric object *O* drawing.

3 For each slice *i*, all or portion of the slice is assigned to the processor **do**

*i*th solidSlice ← CreateSolidSlice (*L*, *i*)

if (The full slice voxelization is done)

Insert *i*th solidSlice into the 3D texture volume *V*

End

End

4 Output the entire volume *V*

End

Figure 5.4: Conventional parallel algorithm for solid voxelization

In both parallel versions, all the slices in the volume space are not voxelized on the same processor. Each processor only voxelizes all or portions of the slices which are assigned to it. However, the multi-core pipelined version does not need to change the sequential code while the conventional version has to change the sequential program to the parallel program as shown in blue color in Figure 5.4. The conventional parallel

programming methods, such as load balancing and scheduling, have to be taken into account. Section 5.4 will further show some experimental results for both of two parallel versions for performance comparison.

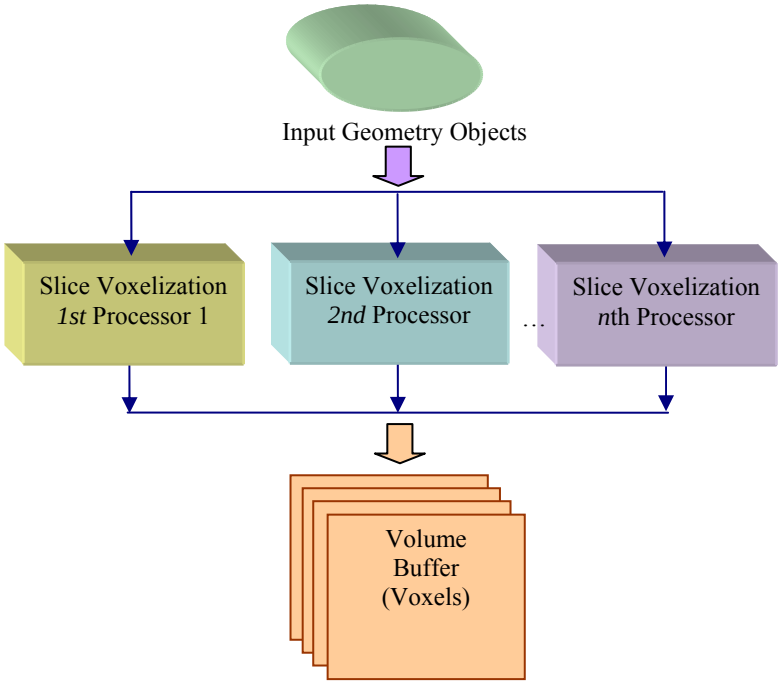


Figure 5.5: Flow chart of conventional parallel solid voxelization

5.3 Principle of Parallel Solid Voxelization by Multi-Core Pipeline

The principle of parallel voxelization by the multi-core pipeline is described in Figure 5.6. In the figure, each step will go through the entire multi-core pipeline. For each object, the next step cannot start until the previous step is finished. For both Step 1 and Step 2, the outputs need to be put into the memory in the multi-core system so that they can be loaded into the next stage, as inputs for the next stage. Step 1 and Step 2 are indispensable for voxelization.

In Step 1, the input data will be a stream of polygons which construct the geometric objects. As soon as the first GPU core is filled it can start processing the received block of data. Upon an interrupt, the contents relevant to the first GPU core is switched to the second GPU core while the first GPU core starts to receive the subsequent data stream. At this time, the second GPU core continues the data processing which the first GPU core stops. The output will be a display list for drawing the geometric objects. After step 1, all relevant object data and display lists will be retrieved by the same program in all the GPU cores.

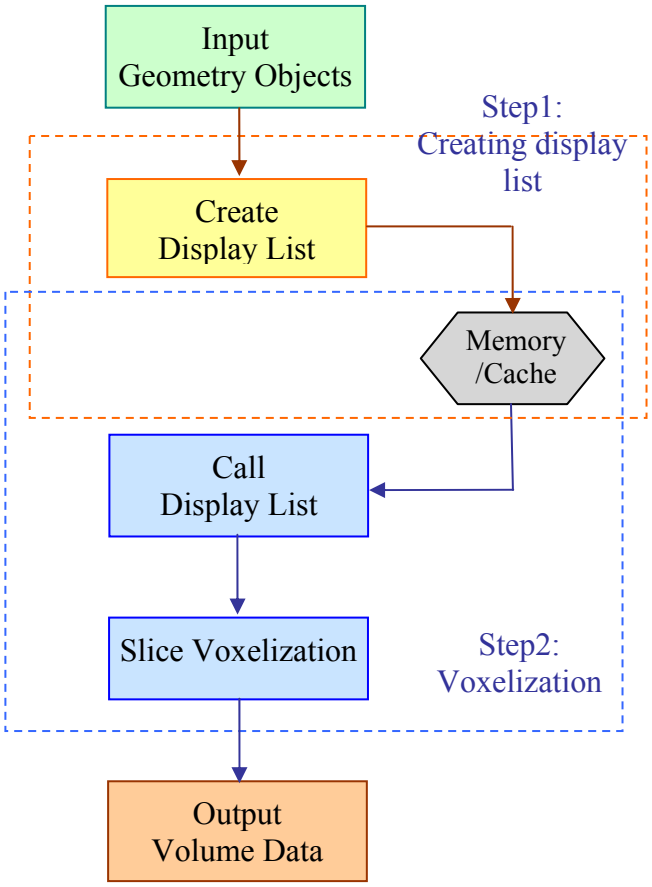


Figure 5.6: The principle of parallel voxelization by multi-core pipeline

In Step 2, beside the geometry object data and display list, the incoming data includes the number of the slice. Upon an interrupt, a GPU core will switch its memories, including framebuffers, and caches to the next GPU core. Especially, the outcomes in the framebuffers including stencil buffers and depth buffers after the processing relevant to this core will be switched to the next core correspondingly. The GPU core will also pass the program status word so that the next core can resume operations where the previous core left off.

5.4 Experimental Analysis

I developed a function simulator to evaluate this multi-core pipelined parallel solid voxelization system and compare it with existing multi-core GPU systems. In this section, I ran both sequential and parallel solid voxelization algorithms described in this dissertation on this simulator.

5.4.1 The Performance

Let us consider the time complexity of solid voxelization on one single-CPU system with one GPU in Equation 3.7 and the general time complexity on one conventional processor in Equation 4.1. It clearly shows that Equation 4.1 can be used for representing the time complexity of the solid voxelization algorithm in another fashion. Thus, for the N-core pipelined parallel voxelization system, the speed-up of the execution time on the new *N*-core pipeline system compared to the time on one single-CPU system is still the same as Equation 4.10

$$Speedup = \frac{N}{1 + \frac{(N-1)m}{n} + \frac{(\left\lceil \frac{n}{m} \right\rceil + N - 2)d}{n}} \approx \frac{N}{1+d} \tag{5.1}$$

where m is the number of cycles of the Processing operation. Clearly, when the number of cores N increases, the speed-up or performance increases correspondingly. That is, the execute time on the N-cores pipelined system decreases as N increases.

5.4.2 Experimental Results and Comparison

The experimental results of the real-time solid voxelization are reported in the following figures.

Figure 5.7 and Figure 5.8 show the experimental results on the single-GPU system using the real-time solid voxelization algorithms described in Figure 3.5 and Figure 3.6. Figure 5.7 indicates the each slice voxelization result from slice 1 to slice 16. Figure 5.8 shows the time spent on each slice voxelization correspondingly.

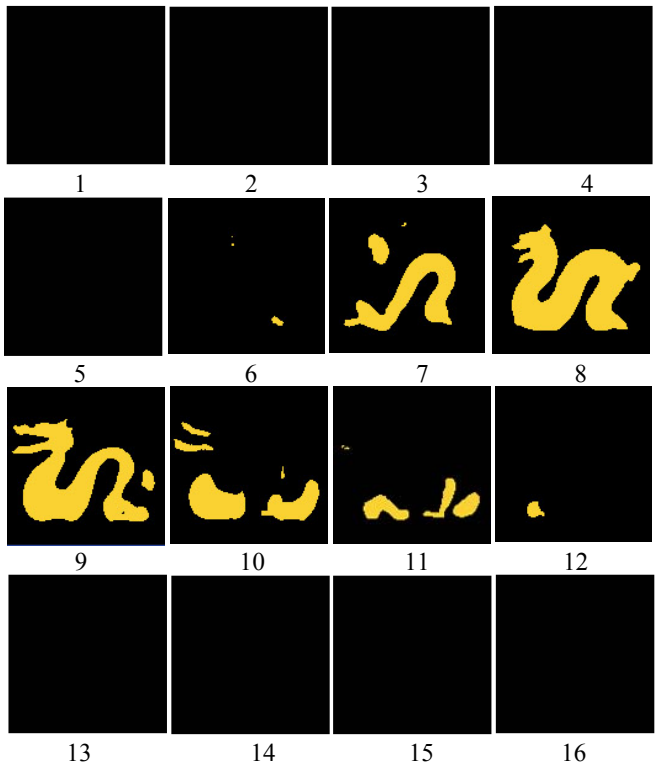


Figure 5.7: 16 slices of dragon_res4 model after voxelization (128x128x16)

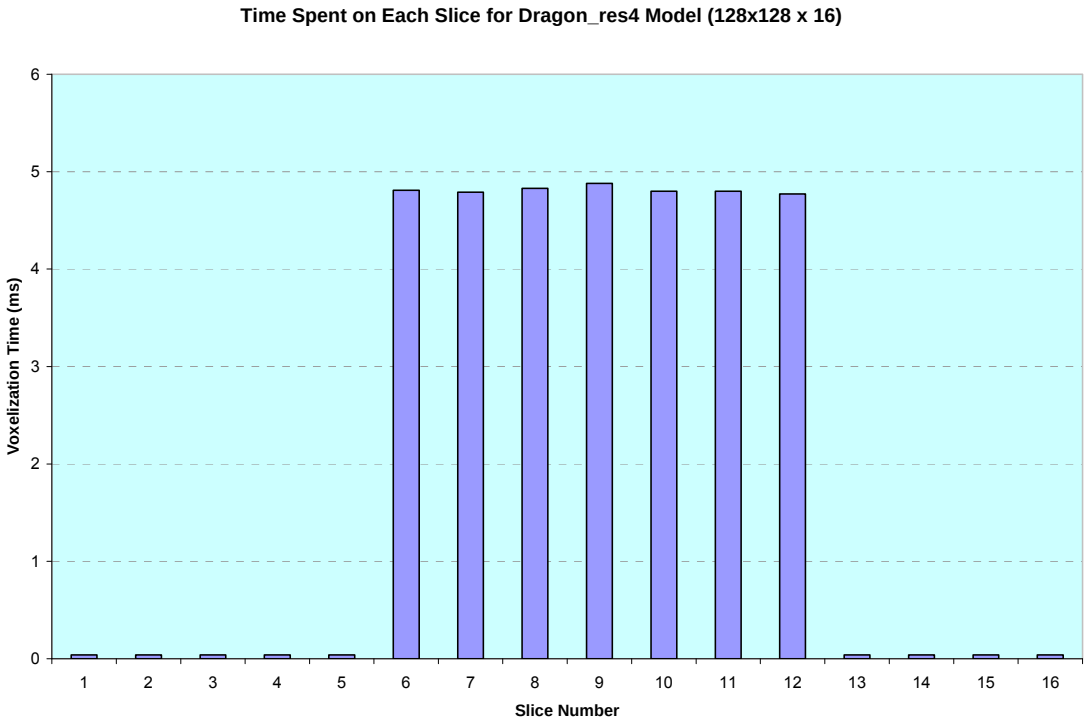


Figure 5.8: Time spent on each slice voxelization for dragon_res4 model (128x128x16)

Figure 5.9 indicates the performance of the multi-core pipelined GPU system for solid voxelization varying the number of cores and switching time. The same sequential solid voxelization program as shown in Figure 5.7 was run on the multi-core pipelined parallel system. Note that forced interrupts were employed for switching control inside the system. It clearly shows the performance increases with the increase of the number of cores. When the switching time by forced interrupts increases, the performance will decrease.

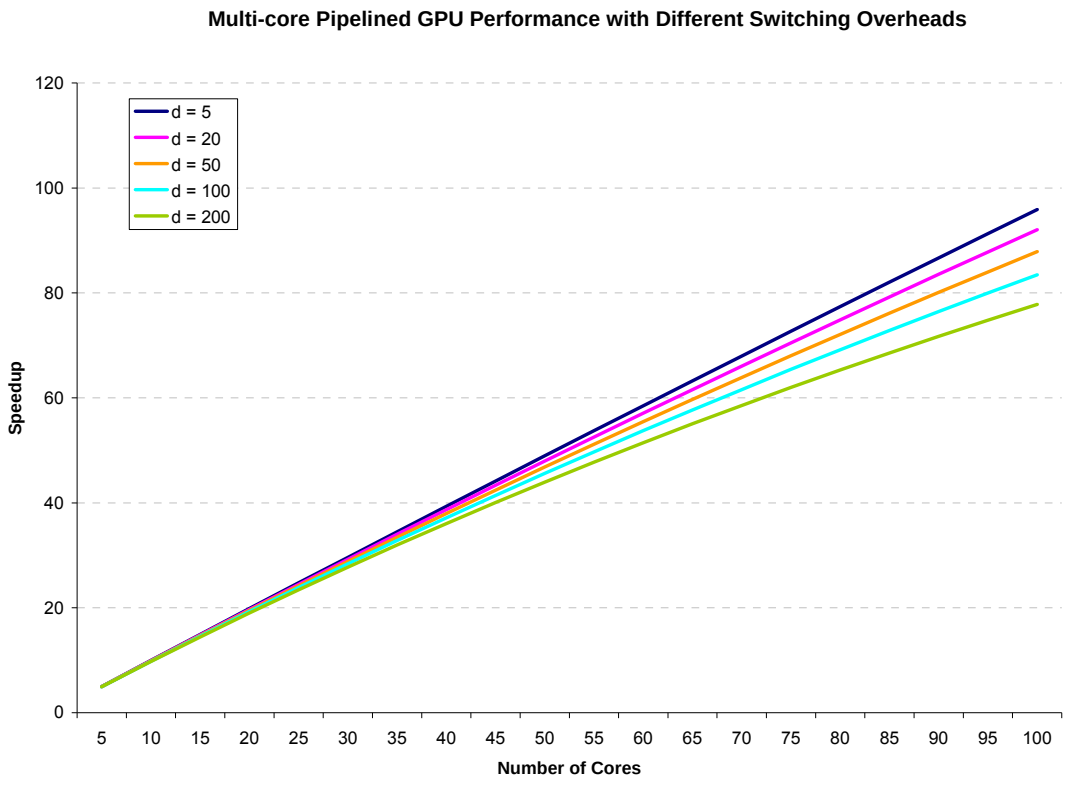


Figure 5.9: Multi-core pipelined GPU performance with different switching overheads

Figure 5.10 shows the comparison of the new multi-core pipelined GPUs and existing multi-core GPUs to voxelize the same geometric model into the same model with the size of 128x128x16. It obviously indicates the performance of new multi-core GPU architecture is better than existing ones. This is because data partition and load balancing and scheduling have to be considered for existing multi-core GPU systems. Moreover, these conventional parallel methods cannot be done easily on current multi-core systems. The performance can be affected by different data partition or load balancing and scheduling methods. However, the new multi-core pipelined GPU systems do not need to consider the data distribution and load balancing issues. All of them will be automatically scheduled onto the different cores. Therefore, the sequential program can be used without any change for parallel applications.

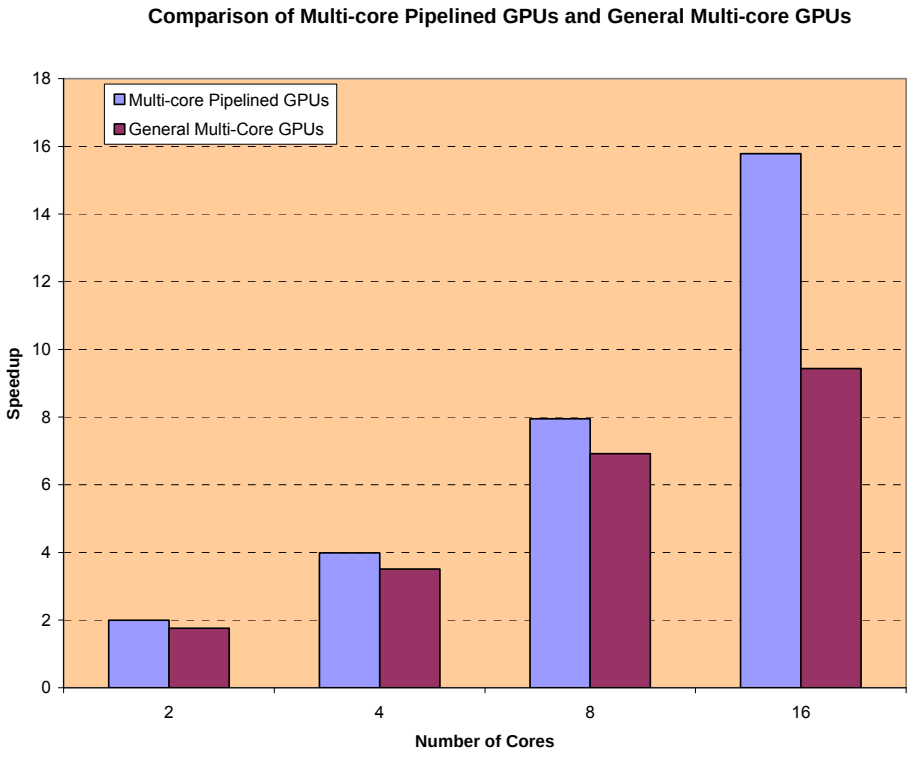


Figure 5.10: Comparison of multi-core pipelined GPUs and existing Multi-core GPUs

5.5 Summary and Advantages

This dissertation presents a multi-core pipelined parallel system for real-time solid voxelization based on the new multi-core pipelined GPU architecture. The proposed system provides a simple and effective implementation for on-the-fly parallel graphics and visualization by switching the entire data from processor to processor. It does not need conventional complicated parallel computing methods, such as load balancing, scheduling, etc. This is exactly the advantage of this proposed parallel system.

To summarize, there are several overall advantages of this system as follows:

- provides the continuous data processing of intensive information flows
- only requires practically the same software as ordinary sequential solid voxelization algorithm
- avoid the data duplication by using a dynamic memory/cache management using crossbar switch
- reduces large bus traffic using the bus crossbar
- avoids the need for synchronization among the processors
- avoids busy waiting of processors on a spin-lock
- eliminates the data dependencies by using slice-independent voxelization algorithm
- the time complexity of this parallel algorithm only affects the length of the pipeline
- be efficient, robust, flexible, easy to implement, and convenient to integrate into other applications

6 Case Study 1: Volumetric CSG Modeling

CSG modeling using volumetric techniques is a new and promising approach. Due to the need for large volume data processing, however, it is difficult to perform interactive volumetric CSG (VCSG) modeling. Since volume techniques are used, volume data sets can be naturally included in the CSG construction process. Such CSG models allow solid modeling and CAD/CAM applications to input objects from sampled or scanned datasets in the design and manufacturing processes. Also, they are potentially useful for other applications involving both geometric objects and volume datasets, such as surgical simulation and biomedical data exploration.

In this chapter, a fast VCSG modeling approach is presented using real-time solid voxelization proposed in this dissertation. The VCSG modeling algorithm builds a new pipeline for volume modeling by slice sweeping. It is a significant improvement over the prior work on CSG model voxelization as described in [Fan98][Fan00]. Furthermore, the algorithm makes use of commonly available graphics hardware features, and is able to provide interactive feedbacks for the modification and manipulation of generic CSG models. Thus, it can be an effective tool in interactive CSG modeling and volume graphics environments.

In the following, CSG model and Volume Scene Tree (VST) are respectively introduced at first. The VST operators are given in mathematical functions. In Section 6.3, an important technology for volume modeling -- volume scene computation based on

slice sweeping, and corresponding algorithms and pipeline are discussed in detail. In Section 6.4, the volume scene tree pruning technique is discussed in detail. Section 6.5 gives some experiment results at interactive speed and comparisons. Also, it provides different experimental results for difference applications, such as CAD/CAM, GIS, and medical application. At last, section 7.6 concludes this approach with a summary of main ideas.

6.1 CSG Model

Let a scalar volume *V* be represented by an intensity function *I(P)*, where *P* is sampled over a 3D regular grid in a volume space. A solid object can be extracted from a volume by thresholding with an intensity interval *[s; t] (s > t).* More specifically, a point *P* belongs to the object if and only if $I(P) \in [s;t]$. This process may also be defined as a *thresholding operation* $F_{[s,t]} : V \rightarrow R^3$, i.e.

$$F_{[s,t]}(V) = \{P \in R^3 : I(P) \in [s,t] \ \} \tag{6.1}$$

Multiple intensity intervals may also be used in the thresholding operation. The result is the union of the objects defined by the individual intensity intervals:

$$F_{\bigcup [s_i,t_i]}(V) = \{P \in R^3 : I(P) \in \bigcup [s_i,t_i] \ \} \tag{6.2}$$

In general, a CSG model can be constructed by the following two types of primitives:

1. A general solid model defined by its boundary surfaces.

2. A solid object defined by a thresholding operation from a volume data set.

The thresholding operation provides additional flexibilities through the changes of the thresholding intervals to generate objects from different layers of the volume. This can be considered part of the CSG editing process, and is an essential feature for the designing

of volume based objects. It should be mentioned that although volume data sets are used, the Boolean operations are only carried out with the (binary) solid objects extracted from the volume data. An example of such CSG models is given in Figure 6.1.

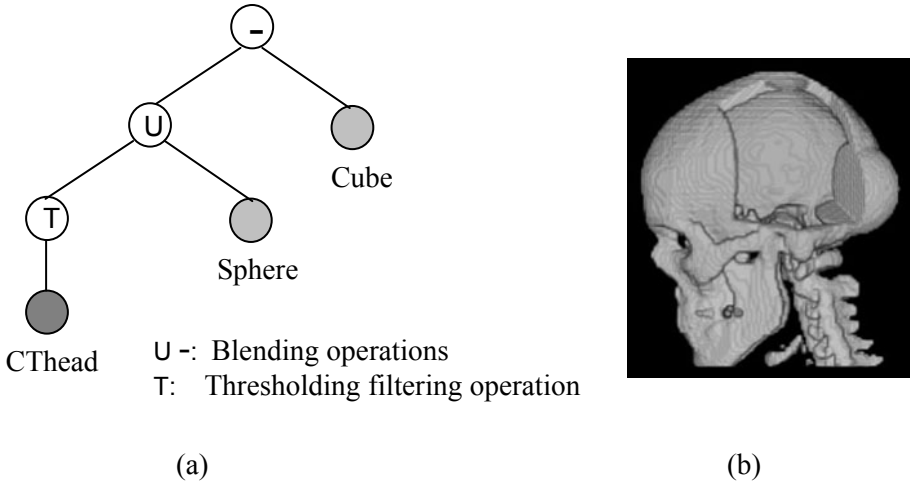


Figure 6.1: A CSG tree example using a CT dataset
(a) The CSG tree (b) The image result

The binary voxelization of a CSG model is the process of generating a binary volume of the CSG model within a given volume domain (region of interest). Only 0 and 1 intensity values are generated in binary voxelization, with 1 representing an inside point and 0 outside point. The result from the binary voxelization can be directly rendered, as a solid object, by 3D texture mapping. It can also serve as a volume space mask for the volume rendering of the CSG model using the original data sets. If the binary volume is considered as a spatial enumeration representation of the CSG object, it can also be used in many other solid modeling applications such as integral property computation, finite element analysis, and layered manufacturing.

6.2 Volume Scene Tree

6.2.1 Definition of Volume Scene Tree

In order to emphasize the difference from previous definitions related to volumetric CSG modeling, and to obtain more efficient interaction in practice, we use a non-binary tree with more operator functions to represent a volume scene, (i.e., extended VCSG models), called *Volume Scene Tree (VST).*

In a VST, each leaf node represents an input dataset or synthetic geometric model, and each interior node represents an operator such as blending or filtering. Blending operators include all bitblt and voxblt Boolean operations. Filtering operators mainly include thresholding and other image processing operations. In addition, an operator could be other operations or functions.

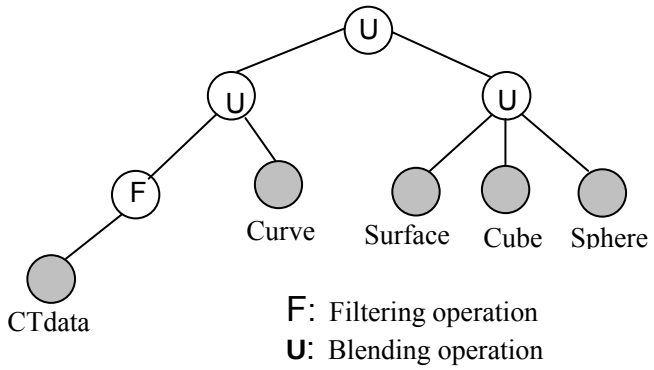


Figure 6.2: An example of a volume scene tree

Figure 6.2 shows an example of such a volume scene tree. In any VST, an interior operation node may have more than two children. It is different from the interior node in a standard CSG tree, which is a binary tree.

6.2.2 Operator Functions

Volumetric Boolean Operations

A volumetric CSG model provides a constructive way of designing 3D models using intermixed geometric and volumetric objects. Several publications such as [Wan94][Fan98][Che00] have given the definition of the volumetric CSG tree and volumetric Boolean operations.

The standard definition of volumetric Boolean operations is straightforward and has been discussed in [FanH00][Wan94]. They can also be directly represented by frame buffer blending functions provided in a 3D graphics API such as OpenGL.

More specifically, a frame buffer blending function in OpenGL, when enabled, is applied between the incoming pixel (source) color and the pixel color at the corresponding location in the frame buffer (destination). Let C_s denote the incoming pixel color (source color) and C_d denote the corresponding pixel color in the frame buffer (destination color). And C is the output color to be written back to the frame buffer. Thus, volumetric Boolean operations in the volumetric scene tree can be implemented with the replacement of blending functions in the scene expression by the following combinations:

Union:

$$C = \max(C_s, C_d)$$

Or:

$$C = k_s C_s + k_d C_d$$

Where k_s and k_d are their corresponding factors, respectively.

Intersection:

$C = \min(C_s, C_d)$

Difference:

$C = \min(C_d - C_s, 0)$

Where, C is equal to 0 if $C_d - C_s < 0$.

VST Operator and Implementation

In a VST, interior nodes represent various operators, such as filtering and blending operations. Basically, volumetric Boolean operations in the volumetric CSG tree can be represented by blending operations in the VST.

A blending function in a VST is an operator with multiple operands, which combines multiple attribute values from different objects at each point in the volume space, similar to the image composition process in [Por84]. Some typical binary blending functions are:

$v(x, y) = \min(x, y)$: minimum material density;

$v(x, y) = \max(x, y)$: maximum material density;

$v(x, y) = x \otimes y$: bitwise logical operations;

$v(x, y) = t_1 x + t_2 y$: linear combination of densities;

$v(x, y) = (1 - a_y) a_x x + a_y y$: Opacity blending.

A filtering function in a VST is an operator that changes the attribute values of the data, similar to the image filtering function. The typical filtering function can be thresholding, boundary detection, color coding, sharpening, blurring, and other image processing operations [Fan98b].

The design and manipulation of the blending, filtering and transformation operators constitute the main process of data exploration and volumetric interaction. One or more parameters can be defined in each operator, and may be modified dynamically by the users during interaction. In fact, users can also define any blending operator except all bitblt and voxblt Boolean operations. Some certain design of the scene expression may also achieve computational tasks beyond visualization.

The blending and filtering functions in the VST are normally applied directly to the slices. This process may also be efficiently implemented using many of the frame buffer blending functions and convolution functions defined in a graphics system. As described previously, combinations of GPU frame buffer blending functions can be used to implement Boolean union, intersection and difference operations. Frame buffer pixel mapping may also be used to implement binary CSG operations. Also, the color table is used as a filtering operator to apply to the frame buffer. In addition, the frame buffer window can often be used as the top slice in the slice stack and directly operated by a filtering or blending operator.

6.3 Volume Scene Computation

A complex volumetric scene can be constructed and composed using various types of geometric and volumetric objects such as curve, surface, solids and CT dataset. Thus how to convert such a complex scene into a unified volume model is very important for volume modeling. This process is called *volume scene computation*, which is carried out by a *slice-sweep* algorithm.

6.3.1 The Principle of Slice Sweeping

Space-sweep (or plane-sweep in 2D) approach has been widely used in many computational geometry and computer graphics problems. This approach can be applied to compute volume information of a scene expression on a 2D slice, which moves across the volume space in regular increments. So this approach is called *slice sweeping*.

The basic idea of slice sweeping is to generate a slice for each object in the scene first, and then apply the blending and filtering functions on the slice in the order defined by scene expression. Before computing the slices, all affine transformations have been concatenated to the leaf level of the expression tree so that data and geometric primitives can be correctly computed during the slicing process. To ensure the correct operational order, a postfix expression will be generated by a standard post-order traversal of the volume scene tree. For example, in Figure 10.2, the expression of the volume scene tree

$$((F(CTdata)) \cup (Curve)) \cup (\cup((Surface),(Cube),(Sphere)))$$

Will be rewritten into a postfix form:

$$(F(CTdata))(Curve) \cup (Surface)(Cube)(Sphere) \cup \cup$$

This algorithm proceeds by moving a slice plane, which is parallel to the projection plane, with a constant step size in a front-to-back or back-to-front order in volume space. The process is shown in Figure 6.3. For each new slice, all primitives are rendered into the slice buffer (i.e., frame buffer) using standard GPU rendering pipeline. This algorithm needs a *slice stack* data structure to store the intermediate result slices. For each slice, stored as a 2D texture image, the algorithm reads one node at a time from the postfix expression of the VST, and pushes slices of leaf nodes (objects) into the stack, and pops

slices from the stack for operator nodes. At the end, the last slice in the stack will be the final evaluated result for this slice.

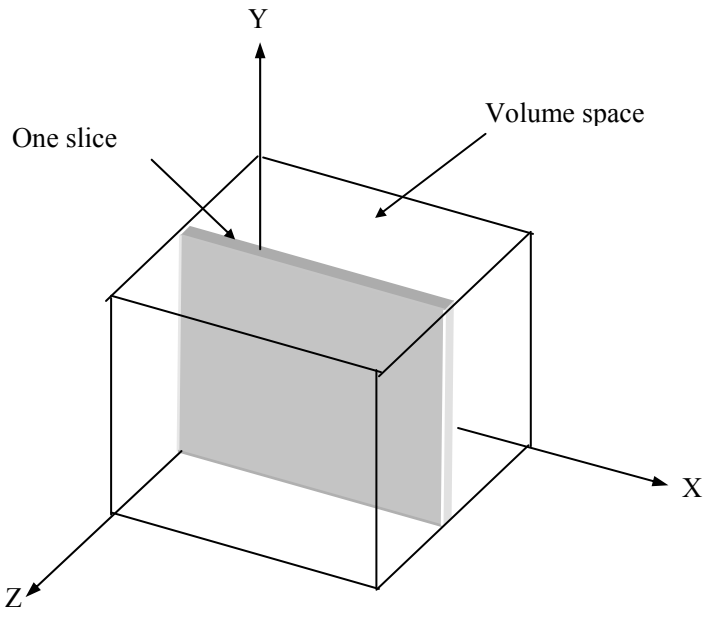


Figure 6.3: Slicing the volume space

6.3.2 Algorithm of Scene Computation

The slice-sweep scene computation algorithm first processes each slice in the volume space by looping with the number of total slices. And then process the VST by tree traversal in post order. If the node in the VST is a leaf node (i.e. an object node), it will be voxelized into the current slice. Otherwise, the node is a filtering or blending function node, and it will be applied to one or more slices popped from the slice stack depending on the function type. If the function node is a blending operator, it can have more than one child node. For this operator node, it needs to pop out k slices from the stack, where k is the number of children for this node. If the function node is a filtering operator, since this

filtering operation defined in volume fusion has the only one operand, so it needs to pop out one slice.

The entire algorithm is described in the following pseudocode:

```
ComputeScene(VST vs_tree)
{
   for ( each slice S) {
       Read the first node P from vs_tree;
       while (P is not NULL) {
          if( P is a leaf node) {
                Voxelize P into the slice S;
                Push (S);
           }
           else if ( P is a filtering operator) {
                S = Pop();
                S = Filter(S, P);
                Push (S);
           }
           else if ( P is a blending operator) {
                for(each operands)
                  T[i] = Pop();
                S = Blending(T[0], …, T[k], P);
                Push (S);
           }
           P = next node in vs_tree;
       }
       S = Pop();
       Output the final slice S into volume memory;
       Move to the next slice;
   }
}
```

In above codes, `vs_tree` represents the VST input by user.

6.3.3 Pipeline of Slice Sweeping

Figure 6.4 illustrates a flow diagram of an idealized version of the slice-sweep scene computation pipeline. In this pipeline, the parts of general OpenGL pipeline are fully used to generate the basic rendered slice for each object so as to achieve interactive scene manipulation.

In the figure, if the node in the VST is a primitive (object), it will first enter the top gray frame. This primitive will be rendered into a slice following the operations of a standard OpenGL pipeline: transformation, clipping, projection, and rasterization. After this step, it will be carried out by different voxelization (*slicing*) algorithms according to the type of the primitive. All results of these voxelized slices will be put into the slice buffers, which will be pushed into a slice stack immediately. If the node is an operator, it will skip those standard graphics pipeline steps in the top dashed frame, and go directly to slice stack operations. Then it will carry out blending or filtering operations for popped slices from the slice stack. If the node is not a root node, it will be pushed into the stack again for the next operation. All of nodes in the VST follow the same pipeline one by one in post order. After traversal of the entire tree, the last slice in the stack will be the final evaluated result for this slice, and be output into the volume memory.

The slice stack is a portion of the memory. In practice, texture memory can be used as the slice stack. And the volume memory storing the entire final volume data is 3D texture memory or a memory with 2D texture stacks.

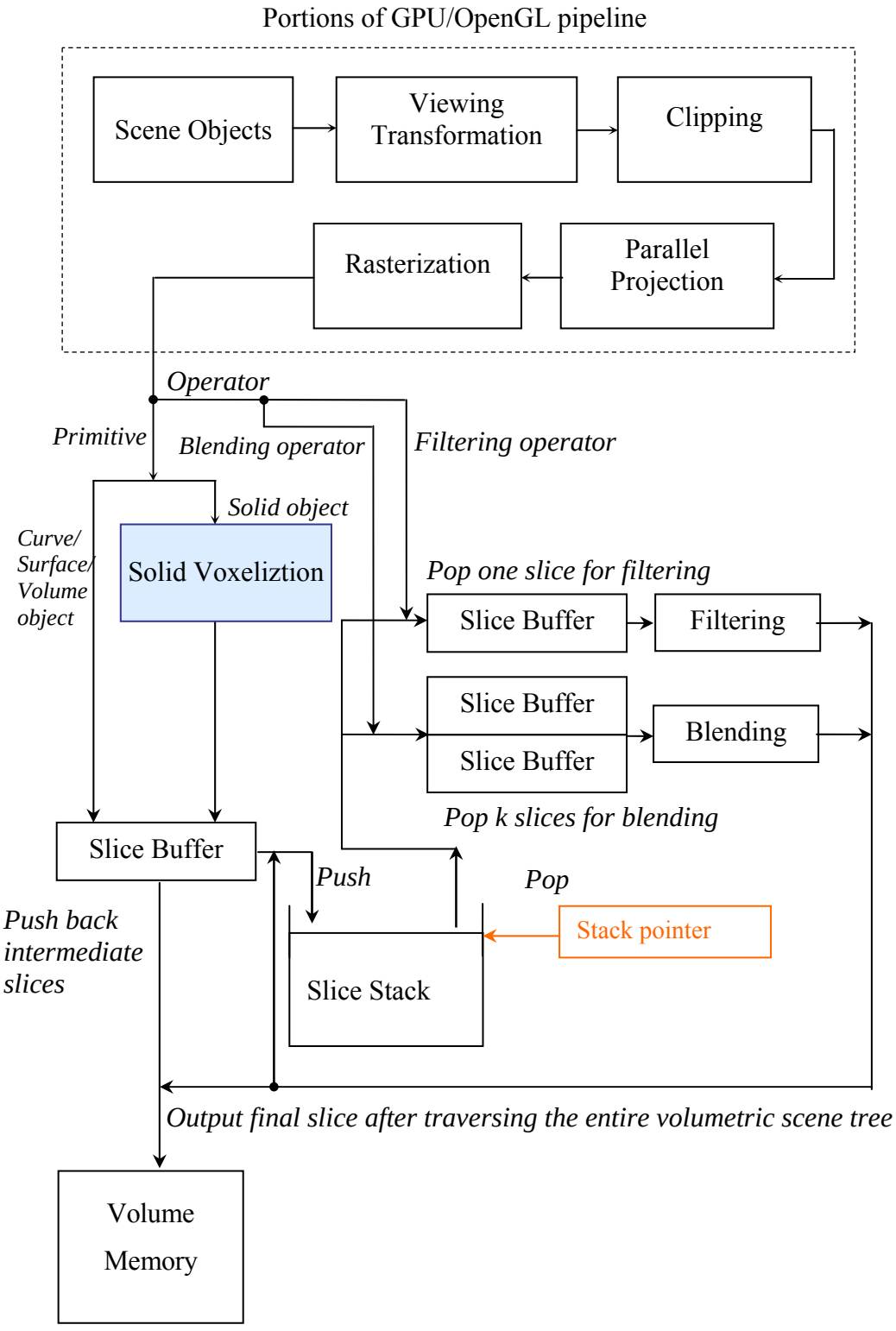


Figure 6.4: The conceptual scene computation pipeline

This algorithm proceeds by moving a slice plane, which is parallel to the projection plane, with a constant step size in a front-to-back or back-to-front order in volume space. The process is shown in Figure 7.3. For each new slice, all primitives are rendered into the slice buffer (i.e., frame buffer) using standard GPU rendering pipeline. This algorithm needs a *slice stack* data structure to store the intermediate result slices. For each slice, stored as a 2D texture image, the algorithm reads one node at a time from the postfix expression of the VST, and pushes slices of leaf nodes (objects) into the stack, and pops slices from the stack for operator nodes. At the end, the last slice in the stack will be the final evaluated result for this slice. Since the GPU rendering result (frame buffer content) is used as the voxelization result, the volume data generated contains true scalar values with color and shading effects.

6.3.4 Slice Computation - Voxelization

In this algorithm, computing a slice with slice-sweeping algorithm starts from each of the leaf node objects in the VST. Since heterogeneous representations are used, this computation process will differ from various object representations. The slicing process is mainly the voxelization of each object type.

Unlike traditional voxelization, which is normally a preprocessing, the voxelization process in scene computation is a dynamic and local (Region Of Interest (ROI)) operation in a volume modeling framework. Therefore, it is important to provide a fast voxelization algorithm for volumetric interactive manipulation. Fortunately, many of the computational operations involved in object slicing (voxelization) can be directly realized by existing features in current 3D graphics systems, which are mostly implemented in hardware.

Slice Computation for Curve and Surface

For Curve and surface objects, a frame buffer window is used as the current active slice. For a given slice, the thin region between the previous slice and the new slice is defined current orthogonal viewing volume, as shown in Figure 10.3, and then all the surface primitives are rendered using standard GPU rendering functions. Since the surface graphics rendering method displays a surface by 2D scan conversion, the intersecting and sampling process are automatically done by the GPU. The system will also use the boundary planes of the viewing volume as clipping planes, i.e. the clipping mechanism of the graphics engine will ensure that only the parts of the surfaces within the slice are displayed.

Slice Computation for Volumetric Data

The slice computation for a regular volume dataset involves data resampling by trilinear interpolation, and is often carried out in a 3D texture mapping environment by GPU. For each slice, the intersecting polygons between the slice plane and the volume data box will be computed. The 3D image coordinates of the intersecting points in the volume will be used as the 3D texture coordinates of the vertices of the polygon, which can then be drawn using 3D texture mapping. The ability to process volume datasets during slicing is important, which allows us to intermix geometric models and volume datasets within a single VCSG modeling process.

Slice Computation for Solid Object

The real-time solid voxelization by hardware acceleration presented in previous chapters is used for the slice computation for a solid object in the VCSG modeling. This

is the key to accelerating the total slice computation due to the complexity and difficulty of the solid voxelization in the VCSG modeling.

6.3.5 Time Complexity

From the view of running time, the total running time *T1* of this algorithm is theoretically *MN*, where *M* and *N* is respectively the number of nodes and slices. However, in [Fan00], two objects in the VST have to be separately converted into their own volumes before they are composted a new volume by a blending operation. So its total running time *T2* is *(M+Mi)N* in theory, where *Mi* is the number of interior nodes. Obviously, *T2>T1* although both of the complexities of two algorithms are *O(MN).* Furthermore, the algorithm in [Fan00] spends large amount of time swapping slices for intermediate volumes in practice.

From the view of occupied memory, this new algorithm needs M slice buffers in the worst case, and only needs two slice buffers in the best case in Figure 6.5. The algorithm in [Fan00] needs *MN* or *2N* slice buffers under the same cases.

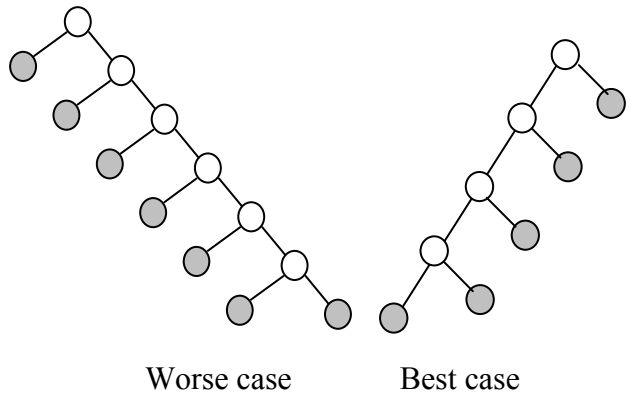


Figure 6.5: Special cases for VST

6.4 VST Pruning

In order to reduce redundant or unnecessary computations, the CSG tree pruning technique can be used. In this algorithm, the pruning process is applied to each slice. For a slice plane, each leaf object will be inside or outside. If outside, the object node and corresponding nodes will be removed in the VST of this slice. This process is different from a single node deletion in a standard tree data structure. Because all interior nodes are operators in a VST, if a leaf node is removed, its parent, the operator could be useless. If so, it should be removed as well. Thus, its sibling(s) could be affected, i.e., moved to the upper level. Each leaf object has its bounding box, which is used to tell if this object is inside or outside. For example, the CTdata in Figure 6.2 is assumed to be outside of a certain slice. The result of the pruned tree of this slice is shown in Figure 6.6.

6.4.1 Definitions of Operators

So far, we have used three basic blending operators: *Max, Min, Subtract*, and one generic filtering operator *F* to describe the tree pruning algorithm. All other further extended operators in VST are handled in the similar way. The basic blending operators are defined in Table 6.4.1. *A, B, and C* represents the nodes.

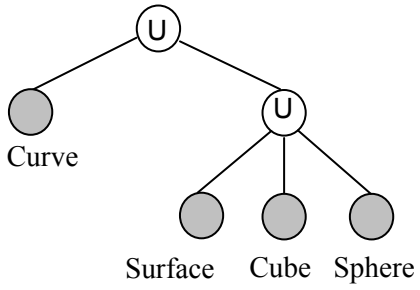


Figure 6.6: A pruned volume scene tree example

Table 6.4.1: Operation definitions

OPERATION	DESCRIPTION
Max(A, B, C, …)	Maximum value among them
Min(A, B, C, …)	Minimum value among them
A - B	$\begin{cases} 0 & (A \le B) \\ A - B & (A > B) \end{cases}$
F(A)	Only for one node

6.4.2 Pruning cases

During the pruning process, if one node is removed, its siblings will have three cases, assuming only the above four basic operators are applied. Let a removed node be set to ϕ. Initially, all outside leaf nodes are ϕ. Now the three cases are summarized in Table 6.4.2.

In summary, Case 0 is the case without any node left after the operation. Case 1 is the case with only one node left. Case 2 has more than one node left. Therefore, most of other current or future operation extensions can follow these cases. The following algorithm we will present may also work for other extension operations.

Table 6.4.2: Three cases for VST pruning

CASE	OPERATIONS	HOW TO HANDLE
Case 0	Min(A, B, ϕ)= ϕ ϕ - A = ϕ F(ϕ) = ϕ	Set current node to Φ. That is, remove it. This case also includes the case that current node is still Φ when all children are Φ no matter what its operation is.
Case 1	Max(A, ϕ)= A A - ϕ = A	Replace current operator node with its child A.
Case 2	Max(A, B, ϕ) = Max(A, B)	Keep this operator node when it has more than 2 children as Max operator. In fact, continue.

6.4.3 Pruning algorithm

The algorithm of VST pruning is described as follows:

```
TreePruning(VST vs_tree)
{
    While(node P is not NULL)
    {
        If(P is operator node)
        {
            According to its children and its operation,
            get one of three cases in Table 9.1.

            if(Case 0)
                 P = NULL;
```

```
            else if(Case 1)
                 replace P with A;
            else
                 continue;
        }
        P = next node in vs_tree;
    }
}
```

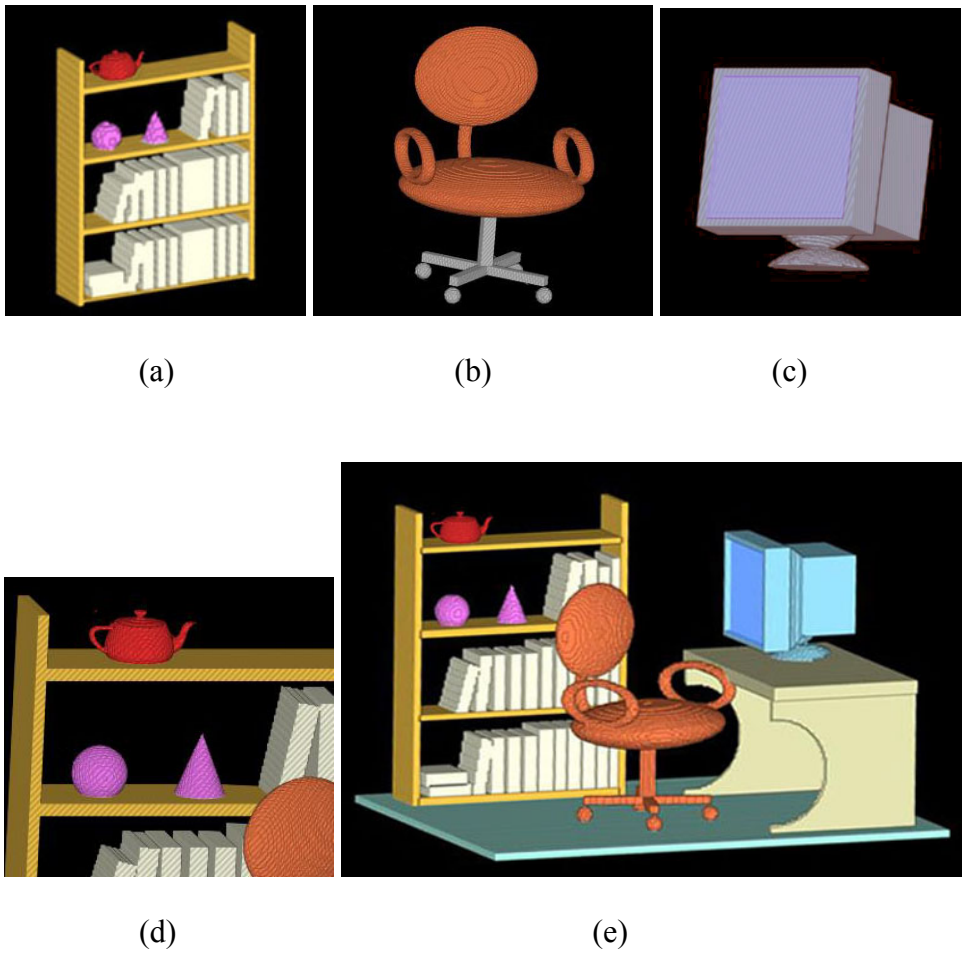
This VST is pruned by post-order traversal.

6.5 Experimental Results

The algorithm of volumetric CSG modeling described in this chapter has been implemented on a Dell PC with Pentium (R) D a single processor 3.0GHZ and 2GB RAM equipped with an Nvidia GTX260 (8-bit stencil buffer, 24-bit depth buffer, 896MB texture memory, and 3D texture mapping supported). The program is written in Microsoft Visual Studio 2005 (VC++ 8.0 /.NET 2.0), OpenGL 2.0, and Cg 2.0. All volumetric data models were rendered using 3D texture mapping.

6.5.1 VCSG Modeling Examples and Time Spent

In Figure 6.7, each VST is computed via slice-sweeping VCSG modeling algorithm and generated into a 128x128x128 32-bit true color volume. Zoom-in effects for ROIs are shown in Figure 6.7 (d).



(a) (b) (c)

(d) (e)

Figure 6.7: Some VCSG modeling examples

Table 6.5.1 shows running time and time comparisons data for VCSG models of Figure 6.7. The results without software optimizations, such as VST pruning and bounding boxes, are also given in comparison to the fast VCSG modeling algorithm presented in this chapter. In general, the VST pruning algorithm is about five times faster on average than the original algorithm. VST pruning further improves the speed by avoiding traveling all nodes in slice stack operations.

Table 6.5.1: Voxelization time for different methods with different models

	TIME SPENT ON SOLID VOXELIZATION (second)			
SIZE	64^3		128^3	
VCSG Model	Non-SW opt.	VST pruning	Non-SW opt.	VST pruning
(a) Bookshelf	0.73s	0.07s	3.48s	0.10s
(e) Chair	0.14s	0.03s	0.42s	0.09s
(c) Monitor	0.13s	0.04s	0.41s	0.10s
(f) Office	0.53s	0.12s	1.64s	0.33s

The primitives used in these examples are composed of mostly polygons represented as OpenGL graphics primitives. Non-polygonal objects include transformed cone, cylinder and sphere objects, and a teapot object. The teapot has a build-in definition in OpenGL using NURBS surfaces. Triangle tessellation is normally used for teapot rendering, which is quite expensive, and may contribute the most in the savings for tree pruning. Mixed datasets and geometric models are shown in Figure 6.8.

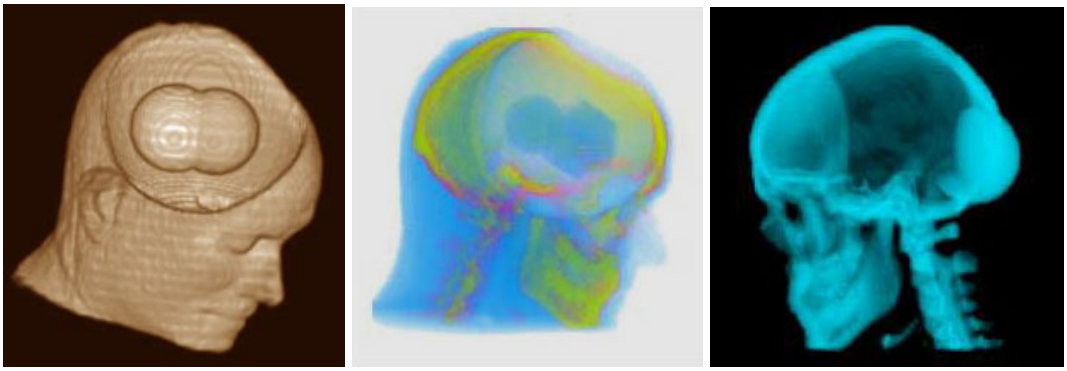
Table 6.5.2: Time comparison of two solid voxelization algorithms for the Office model

Algorithm		64^3	128^3	256^3	512^3
XOR-based		0.59s	2.16s	10.83s	81.92s
New Alg.	Non-SW Opt.	0.53s	1.64s	7.75s	54.22s
	VST Pruning	0.12s	0.33s	1.72s	13.07s

In Table 6.5.2, the new algorithm runs in comparison with Fang and Chen's XOR-based method [Che98, CF00]. Obviously, it clearly shows that the new VCSG modeling algorithm is at least five or seven times faster than the XOR-based VCSG modeling method.

6.5.2 Examples of VCSG Modeling for Other Applications

VCSG modeling can be widely used in various applications, such as CAD/CAM, medical visualization, volumetric segmentation, Geographic Information System (GIS), earthquake, 3D printing, art design, etc. The following experimental results show the examples of different applications using VCSG modeling techniques presented in this chapter.

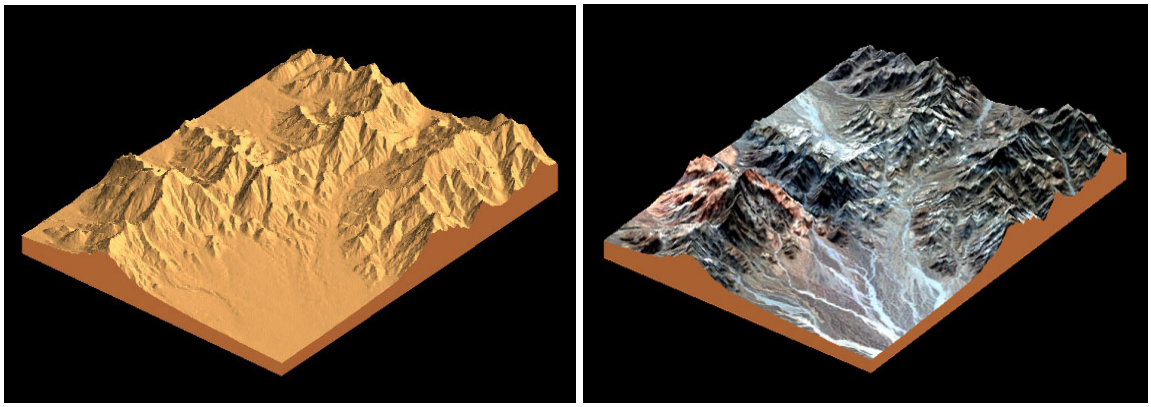


(a) (b) (c)

Figure 6.8: Some VCSG modeling examples for medical application

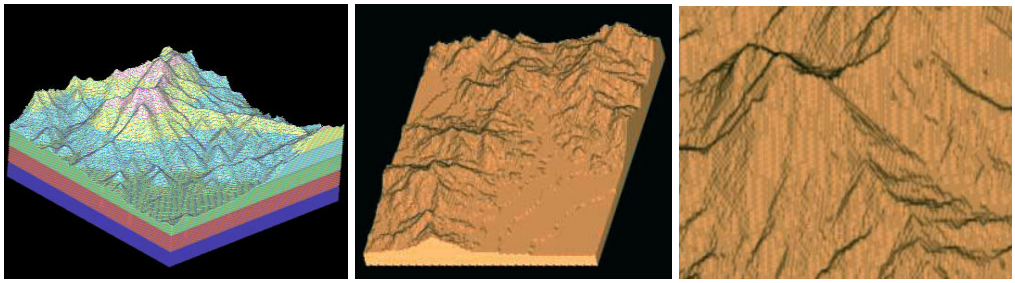
Figure 6.8 gives several examples of medical visualization using VCSG modeling technique. In the figure, (a) and (c) are the VCSG modeling results of CT head and skull datasets mixing with geometric objects to compose the heterirsss VST. (b) is another rendering example of (a) showing interior intensity values from original CT head dataset. It shows this VCSG modeling algorithm can keep the interior intensities of the regular volume dataset during the modeling. Here different colors represent different tissues or bones. This is only a simple example to try volumetric segmentation using VCSG modeling.

Volumetric terrain model is very useful for volumetric GIS, earthquake, and other volumetric applications. During the solid voxelization, several different solid slice functions are employed to generate various volumetric terrains. Figure 6.9 shows several examples of volumetric terrain generation using VCSG modeling technique. In the figure, (a) is the original geometric terrain model (i.e. USGS DEM model) with regular OpenGL 3D surface rendering. (b) is the same DEM terrain model rendering but with remote sensing imagery as a 2D texture mapped on the terrain surface. Image (c), (d), (e), (f), and (g) are the volumetric terrain models generated by different solid slice functions. In (e), (d), (e), and (f), each voxel of the model is assigned to different color based on elevation. Obviously, Image (d) shows the slicing process just starts from the top of the geometric terrain model. So the boundaries of the sliced terrain model are actually the contours. Texture mapping technique is used for volumetric terrain rendering. Using such rendering method and special lighting techniques, the contours are clearly shown as the layers in this volumetric terrain model. In addition, (h) is the ROI of the (g) during interactive VCSG modeling. All of these volumetric data are 128x128x128 sizes.



(a) (b)

(c) (d) (e)



(f) (g) (h)

Figure 6.9: Some VCSG modeling examples for GIS applications

6.6 Chapter Summary

A fast CSG voxelization and modeling algorithm is described in this chapter. To achieve interactive VCSG modeling, this algorithm makes full use of real-time solid voxelization presented in previous chapters in combination with other graphics operations. This algorithm is a significant improvement over some prior work in CSG model voxelization. In particular, a new volume pipeline is used in which each slice is applied to the entire CSG tree, rather than performing a volume-level voxelization for each CSG node. A tree pruning technique is also introduced to provide computational and space optimizations in the CSG modeling process.

Above all, this algorithm makes use of commonly available graphics hardware features, and is able to provide interactive feedbacks for the modification and manipulation of general CSG models. Thus, it can be an effective tool in interactive CSG modeling and volume graphics environments.

VCSG modeling can be widely used in various applications, such as CAD/CAM, medical visualization, volumetric segmentation, Geographic Information System (GIS), earthquake, 3D printing, art design, etc. Some experimental results show the examples of several different applications using this VCSG modeling approach.

7 Case Study 2: Volumetric Collision Detection

A fast volumetric collision detection approach using the real-time solid voxelization technique described in previous chapters is presented in this chapter. In this algorithm, voxelization is the key technique. It improves the collision detection performance as well as detection accuracy based on voxel level.

Collision detection is an important component of many animation, CAD/CAM, gaming, simulation and virtual reality applications [Lin92][Fai94][Bez96]. In traditional collision detection (i.e. polygon-based collision detection [Lin95]), however, since it cannot be determined by checking only the boundaries of the primitives, many complicated mathematical computations have to be employed to detect the collisions for surface model. These algorithms are very time-consuming due to complex intersection operations between points, lines, and planes.

Volumetric collision detection [He97] [Boy99] [Gag00] differs greatly from the traditional collision detection problem. Since volumetric data are only represented by a set of voxels, the intersection operation is only applied between voxel and voxel, (i.e., point and point). Thus the collision detection is simple and fast based on volumetric representation.

This chapter will use a volumetric scene tree (i.e. a VST), to express this procedure of solutions to this problem and use the core real-time solid voxelization technique based on VST for Volumetric Collision Detection (VCD). Section 7.1 describes the color coding

and white pixel check techniques to detect voxel-based collision. Section 7.2 introduces how to design the VST for VCD based on color coding and frame buffer size to make full use of hardware features. In the next two sections, the algorithm of VCD is described in detail and several experimental results demonstrate the effectiveness of this algorithm.

7.1 Color Coding and White Pixel Check

Color coding is used for identifying different objects involved in the scene. The idea of color coding for collision detection is originally proposed in [Boy99, Boy01].

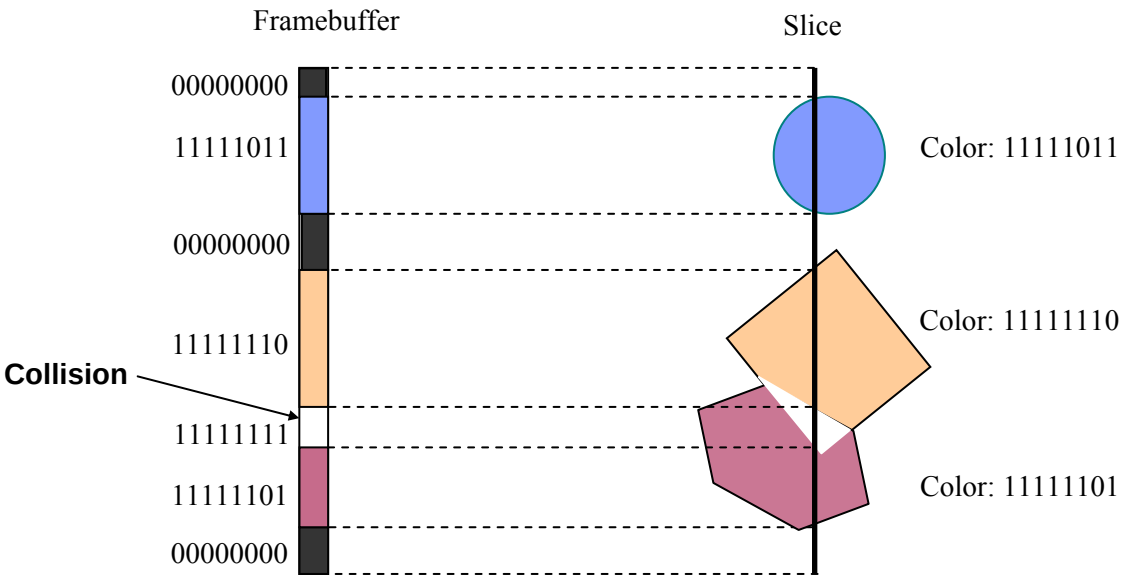


Figure 7.1: Coloring for 3 objects with 1 collision

For an *n*-bit color code, n unique colors are constructed so that the binary representation of each color has exactly one "0" bit and *n-1* "1" bits. The *n* different locations of the "0" bit lead to *n* different colors. By using a logical OR blending function, the resulting color of overlapping objects will become white (i.e. containing all 1's). Then

white pixels are checked if appearing on this slice. The presence of at least one white pixel signifies that a collision has occurred for this slice, as shown in Figure 7.1.

In this chapter, this method of collision detected by white pixel is called *White Pixel Check (WPC)*. *WPC* is an important filtering operation node in the VST. WPC can be implemented on GPU by hardware acceleration. This is a quite different idea to deal with volumetric collision detection from other volumetric collision detection methods.

7.2 VST Design for Volumetric Collision Detection

A root of the VST defined in Chapter 6 for volumetric collision is *WPC*, followed by OR blending operator as his unique child. Theoretically, the bit depth of frame buffer can have any number of bits. However, it is limited by the hardware. Assuming a frame buffer's bit depth is *n*-bit, each OR node in the VST should have at most *n* child nodes, as shown in Figure 7.2 (i.e., 8-bit frame buffer is used), where *WPC* operators also turn all other colors to a new color code for the higher level operations. Notice that, if the node is the CT dataset, it has already been filtered by two filtering functions, thresholding and mapping, which are not shown in the figure. That is, this volume dataset can be thresholded by some value from its orginal volume data, and then mapped into the volume in a certain color with the binary expression. Therefore, the final modeling result is only a binary.

To improve the performance for the VCD processing, the number of interior operator nodes in the VST should be reduced, i.e., VST pruning. In practice, since filtering operator OR and *WPC* have unchanged relationships in the VST, that is, *WPC* always has one OR child, these two filtering operators can be combined into one big filtering node, still call *WPC* node, or *OR+WPC node*.

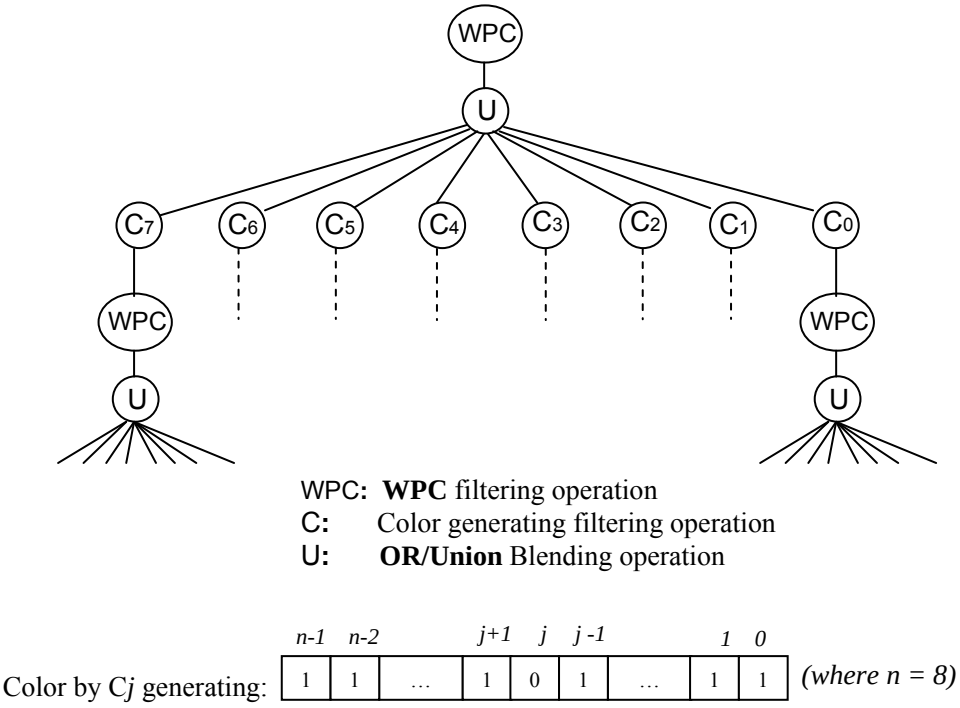


Figure 7.2: A VST for volumetric collision detection

7.3 Algorithm of Volumetric Collision Detection

The entire algorithm of the volumetric collision detection based on slice sweeping is described in the following pseudocode:

```
bool VCD(VST vcd_tree)
{
   for ( each slice S)
   {
       Read the first node P from vs_tree;
       while (P is not NULL)
       {
           if( P is a leaf node)
           {
```

```
                Voxelize P into the slice S;
                Push (S);
            }
            else if ( P is a OR+WPC operator)
            {
                for(each operands)
                  T[i] = Pop();
                S = UnionT[0], …, T[k], P);

                Check if white pixel exists in current slice S.
                if(true)
                   Collision found, return true.
                else
                   Push (S);
            }
            P = next node in vs_tree;
        }
        S = Pop();
        Move to the next slice;
    }
return false;  // collision not found
}
```

In above codes, `vs_tree` represents the VST input by user.

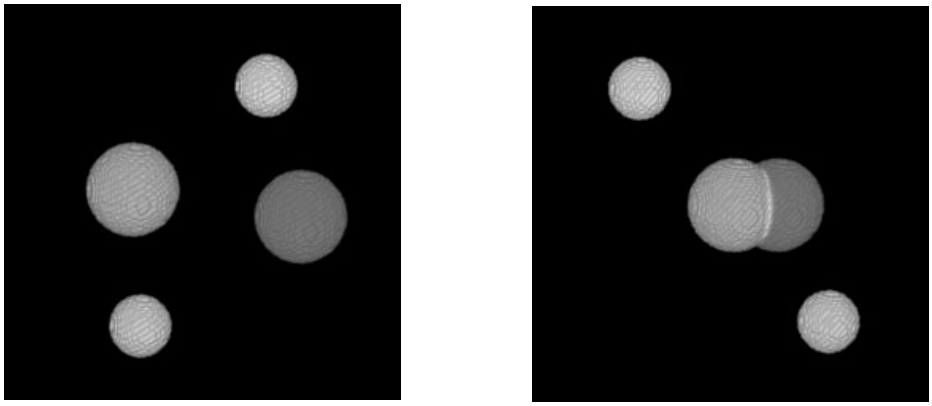
As all other VST operations, the collision detection is performed in a slice-by-slice order (slice sweeping). Thus, both the accuracy and the speed of the algorithm are determined by the number of slices and the slice resolution.

7.4 Experimental Results

In practice, it is very easy to implement this application. After color coding, the only things which need to do are to input a VST of volumetric collision detection as Figure 7.2, and add a new *WPC* filtering function using a single hardware test implemented by the

OpenGL extension, glMinMax(). Once the maximum pixel color of the slice is white, at least one collision between these objects, located in the leaves of the VST, is detected.

There are several application examples shown in this chapter. Figure 7.3 demonstrates a simple example for volumetric collision detection using this method. In the figure, (a) shows no collision occurs, i.e., no white color pixel is found in any slice. It takes less than one second for this process with the volume size of 128x128x128. (b) shows collision has been detected. It is clear to see white color intersection between two big spheres. It only takes very short time to check that collision occurs.



(a) (b)

Figure 7.3: An example of volumetric collision detection
(Image rendering resolution 128x128x128)

In addition, when thresholding is used in volumetric objects, this algorithm allows the collision detection to be selectively applied to different layers of the objects (e.g. bones and tissue), which is particularly useful for virtual surgery applications [Yag97][Lom99], and for implementing haptic force-feedback devices in a virtual environment [Sri97][Dio97] [Mar96].

Figure 7.4 shows some experimental results of a simple example like this. In the figure, the gray bar represents a scalpel. The skull is the result after being thresholded from the whole CT head data. This is a simulation of virtual skull surgery. Image (a) shows the skull has not been touched with the scalpel. But in (b) and (c), we can clearly see those voxels at the intersection will be turned into white colors as long as the scalpel collides some portions of the skull. Thus, the surgeon can know his or her scalpel has touched the skull instead of other portions of his patient's head such as skin or pallium. Also, at the first step, it is necessary to segment skull info out of the CT head dataset by threshold filtering.

Since some OpenGL extension functions are used to quickly check white color pixel and map colors for volumetric collision detection, this application can be implemented on all the graphics systems supporting glMinMax() extension.

The algorithm of volumetric collision detection described in this chapter has been implemented on a Dell PC with Pentium (R) D a single processor 3.0GHZ and 2GB RAM equipped with an Nvidia GeForce GTX 260(8-bit stencil buffer, 24-bit depth buffer, 896MB texture memory, and 3D texture mapping supported). The program is written in Microsoft Visual Studio 2005 (VC++ 8.0 /.NET 2.0), OpenGL 2.0, and Cg 2.0. All volumetric data models were rendered using 3D texture mapping.

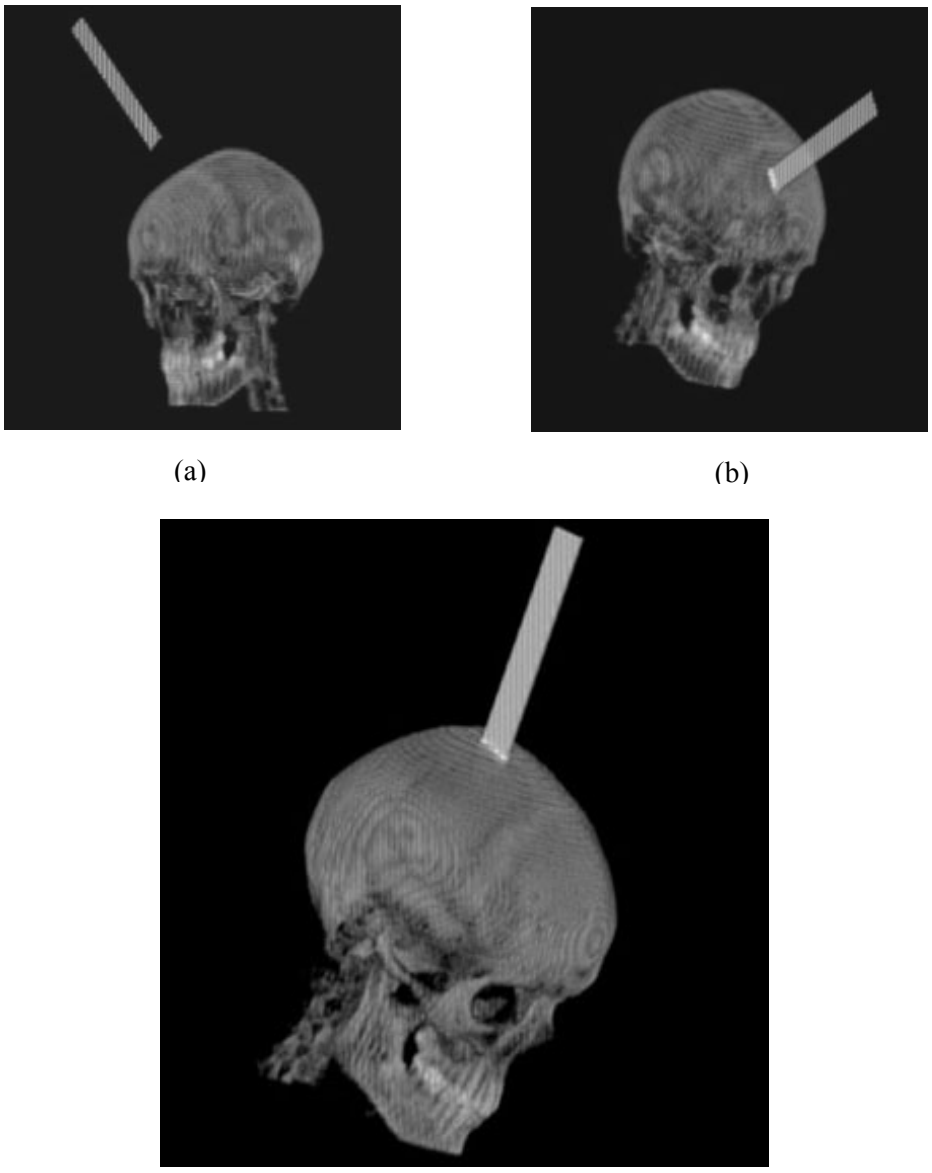


Figure 7.4: An example of volumetric collision detection in virtual surgery application (Image rendering resolution (a) and (b):128x128x128, and (c): 256x256x256)

8 Conclusions

In this dissertation, a new approach to real-time GPU-accelerated solid voxelization is first presented. A novel generic parallel architecture using multi-core pipelining based on program slicing by forced interrupts is proposed and then applied to the parallel solid voxelization system to further improve total performance. The use of this new approach for two important volume applications, volumetric CSG modeling and volumetric collision detection, is also described in detail. In Section 8.1, the algorithms and architetures presented in the previous chapters are further analyzed and the benefits and limitations of this dissertation work are discussed. Some of the possible directions for future work are outlined in Section 8.2.

8.1 Summary

8.1.1 Benefits

All new approaches and architectures developed in this dissertation can not only increase the performance and quality but also reduce the cost. More specifically, several benefits can be gained from this dissertation research as follows:

1) A new real-time solid voxelization approach

This dissertation first presents a real-time solid voxelization approach using dynamic solid slice functions and improved masking techniques based on GPU acceleration to convert solid geometric objects into diverse multi-valued

volumetric models only once. This approach markedly improves not only the voxelization speed for solid geometric objects in real time, but also the voxelization quality with error-free volumetric models. In particular, by dynamically applying different slice functions, any surface-closed geometric model can be voxelized into a solid volumetric representation with any kinds of interior materials, such as rainbow, marble, wood, and so on. This approach has been demonstrated to be effective, robust, flexible, and diverse for both convex and concave complex models. Additionally, this approach is slice based but slice-independent, which is suitable for any parallel processing.

2) A novel generic multi-core pipelined parallel architecture

A novel generic parallel architecture is developed using a multi-core pipeline scheme and a distinct memory and cache dynamic management technology. Through this architecture, most of the existing single-processor software programs can be directly used with nearly no change. This new parallel architecture does not need to cost any extra time and effort for threading, load balancing, scheduling, and data distribution, which are major concerns of conventional parallel programming. Thus, the quality of the new parallel system can also be guaranteed. Furthermore, this new parallel architecture can also be successfully applied to other multi-core/many-core architectures or heterogeneous systems.

3) A new real-time parallel solid voxelization system using multi-core pipelining

To further improve the total voxelization performance for real-time applications, an effective scheme for real-time parallel solid voxelization based on the novel multi-core pipelined architecture is presented. This proposed parallel system

provides a simple and effective solution to real-time solid voxelization by transferring the operating states from core to core. It only requires practically the same solid voxelization software as the ordinary sequential computer described in Chapter 3. This is the most important advantage of this system. Additionally, this voxelization system can be extended to other parallel graphics and visualization applications.

4) A new approach to volumetric CSG modeling using real-time solid voxelization

The case study for volumetric CSG modeling presented in Chapter 6 is one of successful applications using this approach of real-time solid voxelization by GPU acceleration. Chapter 6 describes a new algorithm of fast volume modeling for volume scene by slice sweeping. It allows the intermixing of 3D geometric models and volume datasets in a single CSG modeling process. In particular, a new volume pipeline is used in which each slice is applied to the entire CSG tree, rather than performing a volume-level voxelization for each CSG node. Additionally, a tree pruning technique is also introduced to provide computational and space optimizations in the CSG modeling process. Since each slice is processed independently, parallel slicing is very possible.

5) A new approach to volumetric collision detection using real-time solid voxelization

The case study for volumetric collision detection presented in Chapter 7 is another successful application using the approach of real-time solid voxelization by hardware acceleration. This chapter describes a simple and fast collision detection method based on volumetric representation on the graphics systems. It

improves not only the performance of collision detection but also the detection accuracy based on voxel level.

In summary, the most important advantage gained from using GPU-accelerated solid voxelization approach is the ability to acquire the interior features from the solid geometric objects at interactive speed or even in real time. It is easy to implement and integrate this approach into various interactive applications, such as volume rendering, volume modeling, volumetric collision detection, medical simulation, volume animation, and computer art.

Furthermore, the most important advantage gained from this novel multi-core pipelined parallel architecture is that the same software on an ordinary sequential computer can be directly used for parallel computing as well as reduce a lot of extra time and cost. This is exactly the advantage of this new architecture.

8.1.2 Limitations

However, there are still some limitations or disadvantages in the current approach and corresponding applications as follows:

- The computation of complex solid slice functions, such as Perlin noises, takes much more time than that of color-based solid slice functions.
- The surface boundary of the volumetric object is not generated smoothly during the voxelization in both 2D and 3D spaces. This anti-aliasing problem exists in many voxelization algorithms.
- Due to slice-based voxelization, the running time is influenced by the number of slices.

- In this new multi-core parallel architecture, the same sequential program has to be distributed into each core.

8.2 Future Work

There are several improvements and directions for the future work as follows:

- More efficient methods based on software and hardware acceleration may be explored to improve the computation for complex solid slice functions and quality (i.e. smoothness and accuracy) of the volumetric data during the voxelization. Further improvement may include the use of GPU-based multiple rendering target techniques to voxelize more slices at one time to speed up the total voxelization processing.
- For the new generic multi-core pipelined architecture and the parallel voxelization system using this architecture, the prototype of the architecture and system will be developed based on the principle and design as proposed in this dissertation.
- Both the volumetric CSG modeling and volumetric collision detection use volume scene tree structure and scene computation method. Faster tree traversal, simplification (i.e. tree pruning), slice stack operation (e.g. slice cache) can be further explored to improve the total performance in the future.
- One interesting application area of future research is to extend this approach to volume animation and explore the solid voxelization of time-varying input geometric model data in a progressive manner. Moreover, other practical applications using this real-time solid voxelization approach may need to be

explored in the near future. Especially, volume deformation, 3D printing, and computer art are very promising applications using this technology.

- Another worth mention is that this multi-core pipeline also provides an important solution to processing the continuous intensive information flows without limitation by size. Consequently, this would be very helpful to real-time large-data processing.
- In addition to parallel computing using multi-core pipelining proposed in this dissertation, the idea can also be used for cluster systems using multiprocessor pipelining. But for such systems, the high-speed networking technology may be explored for large-scale applications since processed data may be transmitted to one or many client computers through the network.

At last, we hope that the availability of these new ideas of the fast algorithms and multi-core pipelined architectures will spur more researches in new and innovative interaction techniques as well as multi-core architecture designs, which are still in its infancy and currently there are very few techniques and little expertise available. This may lead to new solutions for difficult parallel computing problems. Moreover, all new approaches and architectures developed in this dissertation can not only increase the performance but also reduce the cost.

References

[Bae00] J. A. Baerentzen, M. Sramek, and N. J. Christensen, 2000, "A Morphological Approach to Voxelization of Solids," *Proceedings Of 8th Central Europe on Computer Graphics, Visualization and Digital Interactive Media*, Pilsen, Czech republic, pp. 44-51.

[Baj00] C. Bajaj, I. Ihm, and S. Park 2000, "Compression-Based 3D Texture Mapping for Real-Time Rendering," *Graphical Models*, *62(6)*, pp. 391-410 (November 2000). Academic Press. ISSN 1524-0703.

[Bau99] K. Baumann, J, Dollner, K. Hinrichs, and O. Kersting, 1999, "A Hybrid, Hierarchical Data Structure for Real-Time terrain Visualization," *Computer Graphics International '99*, IEEE CS Press. ISBN 0-7695-0185-0.

[Bec02] S. Beckhaus, J. Wind, T. Strothotte 2002, "Hardware-based voxelization for 3D spatial analysis," *Proceedings of the 5th International Conference on Computer Graphics and Imaging*. ACTA Press, Canmore, Alberta, Canada, pp. 15-20.

[Ber90] S. Berkovich, Z. Kitov, and A. Meltzer, 1990. "On-the-fly processing of continuous data streams with a pipeline of microprocessors," *Proceedings of the International Conference on Databases, Parallel Architectures, and Their Applications* (PARBASE-90), IEEE Computer Society, Maiami Beach, Florida, March 1990, pp. 85-97.

[Ber92] S. Berkovich, M. Loew, and M. Zaghloul, 1992, “On-Line Processing and Archiving of Continous Data Flows,” *IEEE Proceedings of 35th Midwest Symposium on Circuits and Systems*, Washington DC, United States, pp. 777-779.

[Ber94] E. Berkovich, S. Berkovich, M. Loew, 1994, “A Multi-Layer Conveyor for Processing Intensive Information Flows,” *The Technical Report*, GWU-IIST-94-13, The George Washington University, Washington DC, United States.

[Ber00] S. Berkovich, E. Berkovich, and M. Loew, 2000. "Multi-Layer Multi-Processor Information Conveyor with Periodic Transferring of Processor's States for On-The-Fly Transformation of Continuous Information Flows and Operating Method Therefor", US PATENT No. 6145071, owned by George Washington University. Date issued - November 7, 2000.

[Bez96] H. E. Bez, A. M. Bricis, and J. Ascough, 1996, “A Collision Detection Method with Applications in CAD Systems for the Apparel Industry”, *Computer-aided Design*, *28(1)*, Elsevier Science, pp. 27-32.

[Bit97] I. Bitter, and A. Kaufman, 1997, “A ray-slice-sweep volume rendering engine,” *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (HWWS '97), Los Angeles, California, United States, pp. 121-130.

[Boy99] M. Boyles and S. Fang, 1999, “Slicing-based volumetric collision detection,” *Journal of Graph Tools* 4, 4, pp. 23-32.

[Boy01] M. Boyles and S. Fang, 2003, "3DIVE: An Immersive Environment for Interactive Volume Data Exploration," *Journal of Computer Science and Technology (JCST),* 18(1), pp. 41-47.

[Bre91] D. E. Breen, 1991, "Constructive Cubes: CSG Evaluation for Display Using Discrete 3D Scalar Data Sets," *Porceedings of Eurographics' 91*, pp. 127-142.

[Bre98] D. E. Breen, S. Mauch, and R. T. Whitaker, 1998, "3D Scan Conversion of CSG Models into Distance Volumes," *Proceedings of IEEE/ACM Symposium on Volume Visualization*, pp. 7-14.

[Cab94] B. Cabral, M. Cam, and J. Foran, 1994, "Accelerated Volume Rendering and Topographic Reconstruction Using Texture Mapping Hardware," *Proceeding of 1994 Symposium on Volume Visualization*, pp. 91-98.

[Che98] H. Chen and S. Fang, 1998, "Fast Voxelization of Three-Dimensional Synthetic Objects," *Journal of Graphics Tool*, 3(4):33-45.

[CF00] H. Chen, and S. Fang, 2000, "Hardware Accelerated Voxelization," *Computer and Graphics* 24, 3, pp. 433-442.

[Chi97] T. Chiueh, 1997, "A Virtual Image-Space 3D Rasterization Architecture," *SIGGRAPH / Eurographics Workshop on Graphics Hardware*, ACM Press, New York City, NY, pp. 69-78.

[Cem03] C. Cebenoyan and M. Wloka, 2003, "Optimizing the Graphics Pipeline," *Game Developers Conference* (GDC2003), San Francisco, California, United States.

[Che00] M. Chen, J.V. Tucker, and A. Leu, 2000, "Constructive representations of volumetric environments," *Volume Graphics*, Springer-Verlag, pp. 97-117.

[Coh91] D. Cohen and A. Kaufman, 1991, "Scan-conversion Algorithms for Linear and Quadratic Objects," A. Kaufman, editor, *Volume Visualization*, pp. 280-301.

[Coh95] D. Cohen and A. Kaufman, 1995, "Fundamentals of Surface Voxelization," *CVGIP: Graphics Models and Image Processing*, 56 (6), pp. 453-461.

[Coh97] D. Cohen and A. Kaufman, 1997, "3D Line Voxelization and Connectivity Control," *IEEE Computer Graphics and Applications,* 17(6):80-87.

[Cro08] "Crossbar Switch" on Wikipedia. http://en.wikipedia.org/wiki/Crossbar_switch. 2008.

[CUDA08a] Nvidia, "NVIDIA CUDA Compute Unified Device Architecture Programming Guide (Version 2.1 Beta)", *Nvidia Technical Reports*. Oct. 2008.

[CUDA08b] Nvidia, "NVIDIA CUDA Compute Unified Device Architecture Reference Manual (Version 2.1 Beta)", *Nvidia Technical Reports* Nov. 2008.

[Cul98] D. Culler, J.P. Singh, and A. Gupta, 1998, "Parallel Computer Architecture: A Hardware/Software Approach," *Morgan Kaufmann*, ISBN 1-55860-343-3.

[D3Dlink] Direct3D: http://www.microsoft.com/windows/directx/default.aspx.

[DTX08] "DirectX: Advanced Graphics on Windows". http://msdn.microsoft.com/en-us/directx/default.aspx. 2008.

[Dac00] F. Dachille and A. Kaufman, 2000, "Incremental Triangle Voxelization," *Proceedings Of Graphics Interface 2000*, pp. 205-212.

[Dan70] P.E. Danielsson, 1970, "Incremental Curve Generation," *IEEE Transactions on Computers*, C-19, pp.80-87.

[DeM03] D. E. DeMarle, S. Parker, M. Hartner, C. Gribble, and C. Hansen, 2003, "Distributed Interactive Ray Tracing for Large Volume Visualization," *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, Parallel and large-data visualization and graphics. IEEE Computer Society, Washington, DC, 12.

[Dio97] J. Dionisio, V. Henrich, U. Jakob, A. Rettig, and R. Ziegler, 1997, "The Virtual Touch: Haptic Interfaces in Virtual Environments," *Computers & Graphics*, *21 (4)*, Pergamon Press / Elsevier Science, pp. 459-468, ISSN 0097-8493

[Ebe03] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley, 2003, *Texturing & Modeling: A Procedural Approach*, Third Edition (series in Computer Graphics). The Morgan Kaufmann.

[Eise06] E. Eisemann, and X. DŽcoret.Fast, 2006, "Scene Voxelization and Applications," *Proceedings of ACM Symposium on Interactive 3D Graphics and Games*, Redwood City, California, United States.

[Fan96] S. Fang, R. Srinivasan, S. Huang, and R. Raghavan, 1996, "Deformable Volume Rendering by 3D Texture Mapping and Octree Encoding," *Proceeding of IEEE Visualization'96*, pp. 73-80.

[Fan98] S. Fang and R. Srinivasan, 1998, "Volumetric CSG -- A Model-Based Volume Visualization Approach," *Proceedings of Sixth International Conference in Central Europe on Computer Graphics and Visualization*, pp. 88-95.

[Fan98b] S. Fang, T. Biddlecome, and M. Tuceryan, 2000, "Image-Based Transfer Function Design for Data Exploration in Volume Visualization," *Proceeding of IEEE Visualization'98,* pp. 319-326.

[FC00] S. Fang and H. Chen, 2000, "Hardware Ac-celerated Voxelization," *Volume Graphics*, Chapter 20, pp. 301-315. Springer-Verlag,.

[FanR00] S. Fang, R. Srinivasan, R. Raghavan, and J. Richtsmeier, 2000, "Volume Morphing and Rendering -- An Integrated Approach,". *Journal of Computer Aided Geometric Design*, 17(1):59-81.

[Fai94] K. M. Fairchild, T. Poston, and W. Bricken, 1994, "Effcient Virtual Collision Detection for Multiple Users in Large Virtual Spaces," *Virtual Reality Software and Technology* (VRST'94), pp. 271-286, Singapore, World Scientific Publishing.

[Fei00] E. Feibush, N. Gagvani, and D. Williams, 2000, "Visualization for Situational Awareness," *IEEE Computer Graphics & Applications*, *20(5)*, pp. 38-45. ISSN 0272-1716.

[Fer03] R. Fernando, and M. J. Kilgard, 2003, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley.

[Fer04] R. Fernando. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. The Addison-Wesley.

[FL00] S. Fang and D. Liao, "Fast CSG Voxelization by Frame Buffer Pixel Mapping", *ACM/IEEE Volume Visualization and Graphics Symposium 2000* (Volviz'00), Salt Lake City, Utah, United States, pp 43-48.

[Gag00] N. Gagvani and D. Silver, 2000, "Shape-based volumetric collision detection," *ACM Proceedings of the 2000 IEEE Symposium on Volume Visualization* (VVS '00), Salt Lake City, Utah, United States, pp. 57-61.

[Gol89] J. Goldfeather, J. Molnar, S. Turk, and H. Fuchs, 1989, "Near Real-time CSG Rendering Using Tree Normalization and Geometric Pruning". *IEEE Computer Graphics and Applications*, 9:20-28.

[Gra03] A. Grama, A. Gupta, G. Karypis, and V. Kumar, 2003, "An Introduction to Parallel Computing," *Design and Analysis of Algorithms*: 2/e, Addison-Wesley, ISBN 0-201-64865-2.

[Gre00] G. M. Nielson, 2000, "Volume modeling," Volume Graphics, pp. 29-48. Springer-Verlag.

[Hau02] D. Haumont and N.Warzee, 2002, "Complete polygonal scene voxelization," *ACM Journal of Graphics Tools*, 7(3):27–41.

[He97] T. He and A. Kaufman, 1997, "Collision Detection for Volumetric Objects Visualization," *Proceeding of IEEE Visualization'97*.

[Hof89] C. M. Hoffmann, 1989. *Geometric and Solid Modeling*. Morgan Kaufmann Publishers.

[Hsi05] H. Hsieh, Y. Lai, W. Tai, and S. Chang, 2005, "A flexible 3D slicer for voxelization using graphics hardware," *Proceedings of the 3rd international Conference on Computer Graphics and interactive Techniques in Australasia and South East Asia* (GRAPHITE '05). Dunedin, New Zealand, pp. 285-288.

[Hua98] J. Huang, R. Yagel, V. Filippov, and Y. Kurzion, 1998, "An Accurate Method for Voxelizing Polygon Meshes," *Porceeding of IEEE/ACM Symposium on Volume Visualization*, pp. 119-126.

[Jia97] D. Jian and S. Vijayan, 1997, "Manufacturing Feature Determination and Extraction - Part I: optimal volume segmentation," *Computer-aided Design*, *29(6)*, pp. 427-440. Elsevier Science.

[Jor03] H.F. Jordan and G. Alaghband, 2003, *Fundamentals of Parallel Processing*, Prentice Hall, ISBN: 0-13-901158-7.

[Kar99] E.A. Karabassi, G. Papaioannou, and T. Theoharis, 1999, "A Fast Depth Buffer Based Voxelization Algorithm," *Journal of Graphics Tools*, ACM, 4(4), pp.5-10.

[Kau86] A. Kaufman and E. Shimony, 1986, "3D Scan-conversion Algorithms for Voxel-based Graphics," *Proceedings of 1986 Workshop on Interactive 3D Graphics*, pp. 45-75.

[Kau87a] A. Kaufman, 1987, "Efficient Algorithms for 3D Scan-conversion of Parametric Curves, Surfaces, and Volumes," *SIGGRAPH'87*, volume 21, pp. 171-179, July 1987.

[Kau87b] A. Kaufman, 1987, "An Algorithm for 3D Scan-conversion of Polygons," *Proceedings of EUROGRAPHICS'87*, Amsterdam, Netherlands, pp. 197-208.

[Kau88] A. Kaufman, 1988, "Efficient Algorithms for Scan-converting 3D Polygons," *Computers and Graphics*, 12(2):213-219.

[Kau91] A. Kaufman, 1991, "Volume Visualization," *IEEE Computer Society Press*.

[Kau92] A. Kaufman, 1992, "The Voxblt Engine: A Voxel Frame Buffer Processor," A. A. M. Kuijk, editor, *Advances in Graphics Hardware III*, Apringer-Verlag, pp. 85-102.

[Kau93] A. Kaufman, D. Cohen, and R. Yagel, 1993, "Volume Graphics," *IEEE Computer Graphics and Applications*, 13:51-64.

[Kau96] A. E. Kaufman, 1996, "Volume visualization," *ACM Computing Surveys (CSUR),* Volume 28, Issue 1.

[Kau00] A. E. Kaufman, 2000, "State-of-art in Volume Graphics," *Volume Graphics*. Springer-Verlag.

[Kni99] G. Knittel, 1999, "TriangleCaster. Extensions to 3D-texturing Units for Accelerated Volume Rendering," *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pp. 25-34, ACM Press, Los Angeles, California.

[Lee82] Y. T. Lee and A. A. G. Requicha, 1982, "Algorithms for Computing the Volume and Other Integral Properties of Solids," *Communications of the ACM*, 25(9):635-650.

[Li05] W. Li, Z. Fan, X. Wei, A. Kaufman, 2005, "Flow simulation with complex boundaries," *GPU Gems II*, Chapter 47, Matt Pharr, Randima Fernando, Editor, Addison Wesley Professional, pp. 677-690.

[Lia01] D. Liao, 2001, "Volume Fusion," *M.S. Thesis*, Purdue University.

[Lia02] D. Liao, and S. Fang, 2002, "*Fast CSG Modeling For Volumetric Scene Using Standard Graphics system*," Proceedings of 7th ACM Symposium on Solid Modeling and Application, Saarbrücken, Germany.

[Lia08a] D. Liao, 2008. "*GPU-accelerated Multi-valued Solid Voxelization by Slice Functions in Real Time*", Proceedings of ACM SIGGRAPH International Conference on Virtual Reality Continuum and Its Applications in Industry. Singapore.

[Lia08b] D. Liao and S. Fang, 2008, "*A Volumetric Framework for the Modeling and Rendering of Dynamic and Heterogeneous Scenes*," Geometric Modeling and Processing, Lecture Notes in Computer Science, Springer-Verlag.

[Lia08c] D. Liao, S. Y. Berkovich, 2008. "*A Conceptual System for Parallel Solid Voxelization Using Multi-Processor Pipelining*", Proceedings of ACM SIGGRAPH International Conference on Virtual Reality Continuum and Its Applications in Industry. Singapore.

[Lin92] M. C. Lin and J. F. Canny, 1992, "Efficient Collision Detection for Animation," *Third Eurographics Workshop on Animation and Simulation*, Cambridge, United Kingdom.

[Lin95] M. C. Lin and D. Manocha, 1995, "Fast Interference Detection between Geometric Models," *The Visual Computer*, 11(10), Springer-Verlag, pp. 542-551.

[LMA01] Nvidia Corporation, "Technical Brief: GeForce3: Lightspeed Memory Architecture". *Nvidia Corporation*. 2001.

[Lom99] J. Lombardo, M. Cani, and F. Neyret, 1999, "Real-Time Collision Detection for Virtual Surgery," *Computer Animation '99,* IEEE Computer Society, Geneva, Switzerland.

[McN99] W. A. McNeely, K. D. Puterbaugh, and J. J. Troy, 1999, "Six Degree-of-freedom Haptic Rendering Using Voxel Sampling", *Proceedings of the 26th Annual International Conference on Computer Graphics and interactive Techniques*, *ACM Press/Addison-Wesley Publishing Co.*, New York, NY, pp. 401-408.

[Mok88] W. Mokrzycki, 1988, "Algorithms of Discretization of Algebraic Spatial Curves on Homogeneous Cubical Grids," *Computers & Graphics*, 12(3/4) pp. 477-487.

[Mor90] J. C. Morrison, 1990, "Fast Anti-Aliasing Polygon Scan Conversion," *Graphics Gems.* pp. 76-83, 662-666. Academic Press. Edited by Andrew S. Glassner. ISBN 0-12-286166-3.

[Nad00] D. Nadeau, 2000, "Volume Scene Graphs", *Porceeding of IEEE/ACM Symposium on Volume Visualization*, Salt Lake City, Utah, United States, pp. 49-56.

[Nae95] N. Shareef and R. Yagel, 1995, "Rapid Previewing Via Volume-based Solid Modeling," *Solid Modeling'95*, pp. 281-292.

[Nie00] G. M. Nielson, 2000, "Volume Modeling," *Volume Graphics*, pp. 29-48. Springer-Verlag.

[Nei93] J. Neider, T.Davis, and M. Woo, 1993. *OpenGL Programming Guide*, Addison-Wesley Press, Menlo Park.

[Noo00] F. S. Nooruddin, G. Turk, 2000, "Interior/exterior classification of polygonal models," *Proceeding of IEEE Visualization '00*, Salt Lake City, Utah, United States, pp.415-422.

[OCL08] Aaftab Munshi, "The OpenCL Specification (Version 1.0)". *Khronos OpenCL Working Group*. Dec. 2008.

[OGL05] OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis. "OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 2 (5th Edition)". 2005.

[OGLlink] OpenGL: http://www.opengl.org.

[Pas04] G. Passalis, I.A. Kakadiaris, T. Theoharis, "Efficient hardware voxelization," *Proceedings of Computer Graphics International 2004* (CGI'04). Volume, Issue, 16-19, pp. 374-377.

[Per85] K. Perlin, 1985, "An image synthesizer," *Proceedings of the 12th ACM Annual Conference on Computer Graphics and interactive Techniques* (SIGGRAPH '85) ACM Press, New York, NY, pp. 287-296.

[Per02] K. Perlin, 2002, "Improving noise," *Proceedings of the 29th ACM Annual Conference on Computer Graphics and interactive Techniques* (SIGGRAPH '02). San Antonio, Texas, pp.681-682.

[Pha04] M. Pharr, and G. Humphreys, 2004, *Physically Based Rendering: From Theory to Implementation* (Series in Interactive 3D Technology). The Morgan Kaufmann.

[Pha05] M. Pharr and R. Fernando, 2005, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. The Addison-Wesley.

[Pfi99] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler, 1999, "The VolumePro Real-time Ray-casting System," *SIGGRAPH'99*.

[Rap97] A.Rappoport and S. Spitz, 1997, "Interactive Boolean Operations for Conceptual Design of 3-D Solids," *Computer Graphics*, *SIGGRAPH 97,* Los Angeles, California, pp. 269-278.

[Rap08] RapidMind. "Easily build applications for multi-core." http://www.rapidmind.net/product.php. 2008.

[Req80] A. A. G. Requicha, 1980, "Representation for Rigid Solids: Theory, Methods and Systems," *Computing Surveys*, 12(4):437-464.

[Rui04] A. J. R. Ruiz, R. J. Segura, F. R. Feito, J. R. Miras, C. J. Ogáyar, 2004, "Voxelization of Solids Using Simplicial Coverings," *Proceedings of the 12th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision* (WSCG 2004), Plzen-Bory, Czech Republic.

[Sei08] Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., and Hanrahan, "Larrabee: a many-core x86 architecture for visual computing". In *ACM SIGGRAPH 2008 Papers* (Los Angeles, California, August 11 - 15, 2008). SIGGRAPH '08. ACM, New York, NY, 1-15. 2008.

[Sha95] N. Shareef and R. Yagel, 1994, "Rapid Previewing via Volume-based Solid Modeling," *Proceeding of 3rd IEEE/ACM Symposium on Solid Modeling and Applications*, Salt Lake City, Utah, United States, pp. 281-292.

[Sra98] M. Sramek and A. Kaufman, 1998, "Object Voxelization by Filtering," *IEEE Symposium on Volume Visualization*, North Carolina , United States, pp. 111-118.

[Sra99] M. Sramek and A. Kaufman, 1999, "Alias-free voxelization of geometric objects," *IEEE Transactions on Visualization and Computer Graphics* 3(5), pp. 251-266.

[Sra00] M. Sramek and A. Kaufman, 2000, "VXT: a C++ Class Library for Object Voxelization," *Volume Graphics*, Min Chen, Arie Kaufman and Roni Yagel, editors, ISBN 1-85233-192-5, pp. 119--134.

[Sto98] Stolte, N. and Kaufman, A. 1998. Efficient Parallel Recursive Voxelization for SGI Challenge Multi-Processor System. In *Proceedings of the Computer Graphics international 1998* (June 22 - 26, 1998). Computer Graphics International. IEEE Computer Society, Washington, DC, 66.

[Sto03] A. Stompel, K. Ma, E. B. Lum, J. Ahrens, and J. Patchett, 2003, "SLIC: Scheduled Linear Image Compositing for Parallel Volume Rendering,"

Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics, Parallel and large-data visualization and graphics. IEEE Computer Society, Washington, DC.

[Thi87] W. C. Thibault and B. F. Naylor, 1987, "Set operations on polyhedra using binary space partitioning trees," SIGGRAPH'87, pp. 153–162.

[Uch83] T. Uchiki, T. Ohashi, and M. Tokoro, 1983, "Collision Detection in Motion Simulation," *Computers & Graphics*, *7 ()*, pp. 285-293.

[Var03] G. Varadhan, S. Krishnan, Y. J. Kim, S. Diggavi, and D. Manocha, 2003, "Efficient max-norm distance computation and reliable voxelization," *Proceedings of the Eurographics/ACM SIGGRAPH symposium on Geometry processing*, Eurographics Association, pp. 116–126.

[Wan93] S. Wang and A. Kaufman, 1993, "Volume Sampled Voxelization of Geometric Primitives," *Proceedings of IEEE Visualization'93*, pp.78-84.

[Wie96] T. F. Wiegand, 1996, "Interactive Rendering of CSG Models," *Computer Graphics Forum*, *15(4)*: 249-261-128.

[Woo99] M. Woo, J. Neider, T. Davis, and D. Shreiner, 1999, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Third Edition.* Addison-Wesley.

[Woo80] J. Woodwark and K. Quinlan, 1980, "The derivation of graphics from volume models by recursive subdivision of the object space," *Proceedings of the Conference of Computer Graphics'80*, Brighton, pp. 335-343.

[Yag96] R. Yagel, D. M. Reed, A. Law, P. Shih, and N. Shareef, 1996, "Hardware Assisted Volume Rendering of Unstructured Grids by Incremental Slicing," *IEEE/ACM 1996 Volume Visualization Symposium*, pp.55-62.

[Yag97] R. Yagel, 1997, "Towards Virtual Surgery: Goals and Achievements," *Fifth International Conference in Central Europe on Computer Graphics and Visualization,* Plzen, Czech Republic.

[Zei95] M. Zeiller, W. Purgathofer, and M. Gervautz, 1995, "Efficient Collision Detection for General CSG Objects," *Computer Animation and Simulation '95*, Eurographics. Edited by Demetri Terzopoulos and Daniel Thalmann. ISBN 3-211-82738-2. pp. 66-79.

[Don04] Z. Dong, W. Chen, H. Bao, H. Zhang, and Q. Peng, 2004, "Real-time Voxelization for Complex Polygonal Models," *Proceedings of Pacific Graphics 2004*, Seoul, Korea. pp.73-78.