GL4D: A GPU-based Architecture for Interactive 4D Visualization

CHU, Alan

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

The Chinese University of Hong Kong
October 2010

香港中文大學
大學圖書館系統
23 APR 2013
UNIVERSITY
LIBRARY SYSTEM
THE CHINESE UNIVERSITY OF HONG KONG

Thesis Committee

Professor WONG, Kin Hong (Committee Chair)
Professor HENG, Pheng Ann (Thesis Supervisor)
Professor WONG, Tien Tsin (Thesis Supervisor)
Professor SUN, Hanqiu (Committee Member)
Professor CAI, Yiyu (External Examiner)

Abstract

While our senses and interactions are confined in the three dimensional space we live in, the human intellectual faculty makes it possible for us to imagine higher dimensional space and objects with the aid of visualization. This thesis describes GL4D, a GPU-based architecture for interactive 4D visualization, for producing imageries of objects inside four dimensional Euclidean space. The GL4D visualization platform comprises of utilities for generating and processing 4D objects from equations to tetrahedral mesh and a tetrahedron-based 4D rendering pipeline. The 4D rendering pipeline in GL4D is implemented on top of OpenGL to utilize recent advances in programmable graphics hardware and achieve interactive frame-rate on mainstream consumer graphics hardware.

論文撮要

雖然我們的日常感官和互動受限於三維世界，但是人類的智慧能讓我們想像更高維的空間與物件。本論文描述一個稱為 GL4D 的架構——GL4D 能利用繪圖處理器互動地可視化於四維歐氏空間的物件。GL4D 是一個四維可視化平台，它包括從數學公式產生和處理四維四面體模型的軟體及一個基於四維四面體的成像流程。GL4D 的成像流程是基於 OpenGL 的，故此我們可以利用可編程繪圖硬件的發展來達致於大眾市場繪圖硬件上加速四維成像至可互動可視化的幀率。

Acknowledgments

I would like to take this opportunity to thank my supervisors Prof. Pheng-Ann Heng and Prof. Tien-Tsin Wong for their guidances, patience, understanding and support for my research. I am also indebted to Prof. Andrew Hanson for his assistance and expert review for theoretical background and correctness of rendered images, and also to Prof. Chi-Wing Fu for his assistance in technical and implementation side of my research.

I am fortunate to have colleagues and friends Jacky Chan, Kwun-Kit Lo, Albert Lam, Edith Ngai, Pat Chan, Clement Lee, Mole Wong, Matthew Chung, Tsz-Ho Yu, Charles Siu, May Woo, Yim-Pan Choi, Raymond Pang, Joe Chan, Dong Ni, Jixiang Guo, Justin Yip, Patrick Cheung and Cleave Lam to engage in countless leisurely and academic, sometimes fierce, discussions that are both entertaining and drive my research forward.

Gratitudes also need to be given to administrative and technical staff in the department for their support: Stephen Lai, Fiona Lam, Kim Law, Annie Pih, Yung-Koon Ping, Angus Siu, Temmy So, Calvin Tsang, Tony Wu, Siu-Yee Yik, with special thanks to Ms. A. D. Zee her guidance and insights for personal development and plans.

Last but not least, I would not be what I am today without the endless love and support from my family.

To people mentioned above and countless other unnamed individuals I give my most hearty thanks to.

Contents

1 Introduction **2**

1.1 Motivation . . . 3

2 Background **4**

2.1 OpenGL and OpenGL Shading Language . . . 4

2.2 4D Visualization . . . 6

2.2.1 3-manifold as Surface for 4D Objects . . . 7

2.2.2 Visualizing 4D Objects in Euclidean 3-space . . . 8

2.2.3 The 4D Rendering Pipeline . . . 9

3 Related Work **11**

3.1 General Purpose Processing on Graphics Processing Units . . . 11

3.2 Volume Rendering . . . 12

3.2.1 Indirect Volume Rendering . . . 13

3.2.2 Direct Volume Rendering on Structured Grid . . . 13

3.2.3 Direct Volume Rendering on Unstructured Grid . . . 18

3.2.4 Acceleration of DVR . . . 19

3.3 4D Visualization . . . 22

4 GL4D: Hardware Accelerated Interactive 4D Visualization **26**

4.1 Preprocessing: From Equations to Tetrahedral Mesh . . . 28

4.2 Core Rendering Pipeline: OpenGL for 4D Rendering . . . 29

4.2.1 Vertex Data Upload 30
4.2.2 Slice-based Multi-pass Tetrahedral Mesh Rendering . . . 30
4.2.3 Back-to-front Composition 38
4.3 Advanced Visualization Features in GL4D 38
4.3.1 Stereoscopic Rendering 39
4.3.2 False Intersection Detection 40
4.3.3 Transparent 4D Objects Rendering 42
4.3.4 Optimization 44

5 Results **48**
5.1 Data Sets 48
5.1.1 3-manifolds in $\mathbb{E}^4$—$\mathcal{M}^{3\to4}$ 49
5.1.2 2-manifolds in $\mathbb{E}^4$—$\mathcal{M}^{2\to4}$ 50
5.2 Performance 69

6 Conclusion **71**

7 Future Work **72**

Bibliography **74**

List of Figures

2.1 OpenGL 4.0 rendering pipeline [3]. 5

3.1 The volume rendering pipeline proposed by Levoy [6]. 14

3.2 3D texture mapping technique proposed by Cullip and Neumann [21]. 21

3.3 Volume bounding box for determining entry and exit point of rays casted from viewing plane [23]. 22

3.4 Tetrahedra classification [16]. 23

3.5 Tetrahedra decomposition [16]. 24

4.1 An overview of GL4D. 27

4.2 Two possible ways of decomposing a hexahedral cell into tetrahedra: 6-tetrahedra (top) and 5-tetrahedra (bottom). 29

4.3 4D transformations in vertex shader. 33

4.4 Two possible vertex ordering in a tetrahedron. p_0 is above the paper and p_1, p_2 and p_3 are on the paper. 34

4.5 Two possible intersection between a tetrahedron and a plane: a triangle or a quadrilateral 36

4.6 Steiner surface in divergent stereoscopic rendering. 41

5.1 Hypercube in convergent stereoscopic view. 52

5.2 Hypercube rotation in $\mathbb{E}^4$. 53

5.3 3-torus in convergent stereoscopic view. 55

5.4 3-torus rotation in $\mathbb{E}^4$. 56

5.5 Trefoil knot in $\mathbb{E}^3$ 60

5.6 Open trefoil knot in $\mathbb{E}^3$ before spinning. 61

5.7 Trefoil knot hidden in knotted sphere in convergent stereoscopic view. 61

5.8 Knotted sphere rotation in $\mathbb{E}^4$. 62

5.9 $(1, 2)$-Fermat surface in convergent stereoscopic rendering. . . . 65

5.10 $(1, 3)$-Fermat surface in convergent stereoscopic rendering. . . . 65

5.11 $(2, 2)$-Fermat surface in convergent stereoscopic rendering. . . . 66

5.12 $(2, 3)$-Fermat surface in convergent stereoscopic rendering. . . . 66

5.13 $(2, 4)$-Fermat surface in convergent stereoscopic rendering. . . . 67

5.14 $(3, 3)$-Fermat surface in convergent stereoscopic rendering. . . . 67

5.15 $(3, 4)$-Fermat surface in convergent stereoscopic rendering. . . . 68

5.16 $(4, 4)$-Fermat surface in convergent stereoscopic rendering. . . . 68

5.17 $(5, 5)$-Fermat surface in convergent stereoscopic rendering. . . . 69

List of Tables

1 Summary of mathematical notations 1

2 Summary of notations in the Direct Volume Rendering section . 1

4.1 Lookup table for acclerating marching tetrahedra 37

5.1 Frame-rate (frame per second) of GL4D for different 4D models and different hardware configurations with different numbers of slices. 70

List of Algorithms

1 `glFrustum` setup for the image for left eye. 39

2 `glFrustum` setup for the image for right eye. 40

3 The 4D dual depth peeling algorithm. 45

4 Generation of all vertices for a bounding hypercube in $\mathbb{E}^n$ 46

5 Generation of a hypercube 51

Summary of Notations

Summary of Mathematical Notations

Notation	Description
$\mathbb{R}^n$	n dimensional real vector space
$\mathbb{E}^n$	n dimensional Euclidean space
$\mathcal{M}^{m \to n}$	m dimensional manifold immersed in $\mathbb{E}^n$

Table 1: Summary of mathematical notations

Summary of notations for Section 3.2.2

Notation	Description
$\vec{p}$	A 3-tuple denoting the location of a voxel within the volume data
$x = f(\vec{p})$	the (scalar or vector) value of voxel at location $\vec{p}$
$\nabla f(\vec{p})$	the estimated gradient at location $\vec{p}$
$C(x)$	the color transfer function
$\alpha(x)$	the opacity transfer function
$C'(\vec{p})$	the final color value of voxel at location $\vec{p}$
$\alpha'(\vec{p})$	the final opacity value of voxel at location $\vec{p}$

Table 2: Summary of notations in the Direct Volume Rendering section

Chapter 1

Introduction

While our everyday experience is fundamentally limited by the dimensionality of the space we live in, it is still possible to glimpse into the world of higher dimensionality by using one of the most important intellectual gift to *homo sapiens* — imagination.

We interact with three dimensional objects in a three dimensional world: three parameters—width, height and thickness—are required to fully specify the size of three dimensional objects. Three dimensional objects also have three perpendicular directions of movement—forward or backward, left or right, and up or down. Is it possible for us to imagine a four dimensional objects in a four dimensional world? A four dimensional object will have its size specified by four parameters and it will have four perpendicular directions to travel when roaming in its four dimensional world. While it is popular to think of the fourth axis being the time axis, this is not necessarily the case. In our research we treat all four axes being homogeneous and equal, and together they form the basis of an abstract four dimensional world.

Visualization plays a vital role in helping us to comprehend and imagine what a 4D object would appear before 4D human beings. Visualization of 4D objects, from hand sketches to computer rendering, is the gateway for us to stand in the shoes of 4D human beings and see what they see. Previous attempts had been focused on generating plausible images for what 4D objects

look like in front of 4D human beings, and output from these researches were static images or pre-animated animation sequence of 4D objects.

With the advances and increase in programmability of graphics hardware it is now possible to compute and render 4D objects at interactive frame-rate. These advances encourage us to build an interactive system that would allow users to manipulate 4D objects and provide instant feedback. We have further developed various advanced and novel 4D rendering and visualization techniques on top of the basic 4D rendering pipeline. These new visualization techniques will be tremendously useful for exploring and understanding novel 4D objects.

1.1 Motivation

Abstract thinking and imagination led to the development of high dimensional mathematics and its subsequent application in physics. We believe that our 4D visualization system can have both pedagogical and research usages in the field of science. Our 4D visualization system can be used in education settings to capture the attention of children and teenagers and keep them more interested in science than conventional science education can. Furthermore, teenagers can better appreciate 4D objects by manipulating them in our interactive visualization system. As modern mathematics and physics are becoming increasingly complex and abstract, we hope that, by providing a platform for manipulating 4D objects interactively, we can advance our understanding of these new theories and make further discoveries from the gained knowledge of their geometrical structure.

Last but not least, it is a human endeavor to go beyond what we know, and GL4D fulfills our curiosity by allowing us to look into the abstract world of higher dimension in a way we have never tried before.

And here is where the fun begins.

Chapter 2

Background

2.1 OpenGL and OpenGL Shading Language

The OpenGL API [1] was created when the underlying hardware is less powerful and flexible, as a result the rendering pipeline contains fixed stages and few configurable options. As graphics hardware becomes increasingly powerful and flexible, more options are exposed and made configurable via the OpenGL API. Recently, advances in graphics hardware allowed us not only to configure the fixed pipeline via predefined options, but to program the pipeline directly. This revolutionary transition from a fixed pipeline to a programmable pipeline had created a new research area for utilizing GPU for general computing [2]—GPGPU (General Purpose Graphics Processing Unit). This new research area focuses primarily on harnessing the powerful parallel floating point processing power in GPUs to perform scientific computations and run parallel algorithms.

Graphics pipeline is programmed by shader programs (Figure 2.1). A high level programming language based on the C language called OpenGL Shading Language (GLSL) is used to write shader programs. The two most basic shaders in a modern GPUs are vertex and fragment shader.

Vertex shader replaces the part in fixed function pipeline between vertex data input and primitive assembly (Figure 2.1). It is responsible for transforming vertices by model view and projection matrices. Vertex shader takes

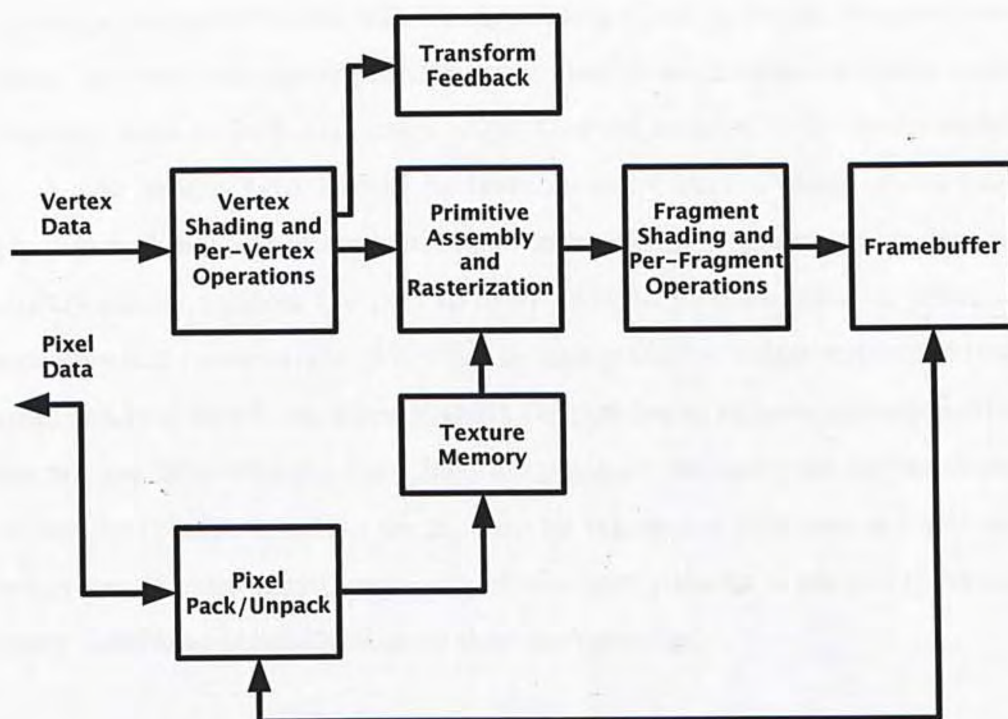

Figure 2.1: OpenGL 4.0 rendering pipeline [3].

one vertex as input, performs all necessary transformation computations and emits one vertex as output. Vertex shaders are invoked to 'shade' every vertices before the vertices are assembled into primitives and rasterized.

Fragment shader replaces the part in fixed function pipeline between rasterization and frame-buffer blending. It is responsible for calculating the color value for each fragment. Fragment shader takes one fragment as input, along with any auxiliary data such as interpolated normal vector and material parameters, computes a color value from shading equation for the fragment and emits the color value as output. Fragment shaders are invoked to 'shade' every fragment from rasterization stage before they are blended to the frame-buffer.

A new shader type had to be invented every time a stage of the fixed pipeline had made programmable. One such example is geometry shader: geometry shader replaces the part in fixed function pipeline between primitive assembly and rasterization. Although geometry shader, unlike vertex and fragment shaders, introduces a new stage to the pipeline to support operations that are not available with the fixed function pipeline: geometry shader can create or destroy primitives within the pipeline by taking one primitive as input and emits zero or more primitives as output. Geometry shader is invoked to 'shade' every assembled primitives before they are rasterized.

2.2 4D Visualization

4D visualization concerns with the problem of visualizing 4D objects, but we first need to define the space where 4D objects live in before we can define the 4D objects themselves. 4D objects live inside four dimensional Euclidean space $\mathbb{E}^4$. An n-dimensional Euclidean space $\mathbb{E}^n$ is defined as a real vector space $\mathbb{R}^n$ with the inner product between vectors $\vec{x}$ and $\vec{y}$ being $\vec{x} \cdot \vec{y} = \sum_{i=0}^{n} x_i y_i$. This inner product definition imposes the Euclidean structure to the real vector space.

The problem of 3D visualization concerns with rendering 2D surfaces forming the boundary of 3D objects. These 2D surfaces are 2-manifolds immersed in 3D Euclidean space ($\mathbb{E}^3$). We define 2-manifolds immersed in 3D Euclidean space as mappings $\mathcal{M}^{2\to3}$ from $\mathbb{R}^2$ to $\mathbb{E}^3$, *i.e.* $\mathcal{M}^{2\to3} : \mathbb{R}^2 \to \mathbb{E}^3$. Analogously, 4D visualization concerns with rendering 3D surfaces forming the boundary of 4D objects. These 3D surfaces are 3-manifolds immersed in 4D Euclidean space ($\mathbb{E}^4$). We define 3-manifolds in 4D Euclidean as mappings $\mathcal{M}^{3\to4}$ from $\mathbb{R}^3$ to $\mathbb{E}^4$, *i.e.* $\mathcal{M}^{3\to4} : \mathbb{R}^3 \to \mathbb{E}^4$.

In this thesis we will use the notation $\mathcal{M}^{m\to n}$ to denote m-manifolds immersed in n dimensional Euclidean space $\mathbb{E}^n$.

2.2.1 3-manifold as Surface for 4D Objects

One way to imagine what a 3-manifold looks like in $\mathbb{E}^4$ is by using the flat-land analogy. The flat-land analogy begins with having point-landers living on one dimensional surfaces, *e.g.* a curve. Point-landers can only move back and forth along a curve, and their movements have only one degree of freedom. Flatlanders, as its name implies, live inside a two dimensional surface, and they have two degrees of freedom when gliding in the 2D surface. Finally, spacelanders, such as we human beings, live within a three dimensional surface and have three degrees of freedom in our movements.

Assuming that there are another form of super-human in a four dimensional Euclidean space $\mathbb{E}^4$, and they encounter a 3D human being on the surface of a 4D object (say, a hypersphere). Although the 3D human being on the hypersphere surface think they have exhausted all degrees of freedom in its movements and the surface they are living in comprises the whole world known to them, 4D super-human outside the hypersphere has access to an additional fourth degree of freedom that is unavailable to the 3D human wandering within the confinement of the hypersphere.

2.2.2 Visualizing 4D Objects in Euclidean 3-space

There are many ways to visualize a 4D object. One of the simplest way is to present a sequence of 3D objects and this sequence of 3D objects, when one is stacked on top of another along an imaginary axis of the fourth dimension, is equivalent to the 4D object in its full glory. It would be easier to imagine this if we start from lower dimensions: a one dimensional line-lander would have difficulty imagining how a two dimensional circle looks like in a two dimensional space, but it can be told that a two dimensional circle is formed by a stack of one-dimensional lines with different lengths; a two-dimensional flat-lander would have no idea how a three dimensional sphere looks like in a three dimensional space, but it is told that a three dimensional sphere is the same as putting a bunch of circles of different radii together; finally a three-dimensional space-lander would have no idea how a hypersphere looks like in a four dimensional space, but it is told that a hypersphere is actually a stack of ordinary spheres with varying radii. Another angle to understand this is that the sequence of the objects actually represents the cross sections of a higher dimensional object, and we can form the concept of the 4D object in question by stacking cross sections of it mentally. While this cross section-based interpretation of 4D objects is simple enough to understand, it fails to provide a holistic view of the whole object at once, making this approach inappropriate for visualizing 4D objects under interactive manipulation.

The second approach for visualizing 4D objects is to simulate the vision system of 4D super-human. We again work our analogy from low dimensions and extend to high dimension for easier understanding. A two dimensional flat-lander has a one dimensional retina that produces one dimensional images of the two dimensional flat-land world; a three dimensional space-lander has a two dimensional retina that produces two dimensional images of the three

dimensional space-land world; finally, by extending the argument, a four dimensional hyperspace-lander will have a three dimensional retina to see the four dimensional hyperspace-land world. This approach of 4D object visualization relies on our ability to simulate and reproduce the retina image that our hypothetical 4D super-human sees when living in a hypothetical four dimensional world. The biggest drawback of this approach is that while a four dimensional hyperspace-lander can directly see every voxel in the three dimensional retina image without difficulties—just like we can see every pixel on a two dimensional image at once without one pixel being occluded by another pixel—a person living in a three dimensional world couldn't do this as some voxels of the retina image occlude other voxels. This problem can be alleviated by allowing space-landers to rotate tune the transparency of the retina image. A space-landers can, by manipulating these two controls, reconstruct a mental model of the full three dimensional retina image. Such a system needs to provide two sets of controls for manipulating the rendering of a 4D object: the first set of controls controls the transformation of the 4D object before projection to the three dimensional retina and the second set of controls controls the transformation of 3D retina image before projection onto 2D screen. This system of two controls allows users to fully comprehend the three dimensional structure of the projected 4D object within the 3D retina.

The second approach to 4D object rendering is chosen in our research because we need a visualization approach that provides a holistic view of 4D objects while being manipulated under heavily interactive use cases.

2.2.3 The 4D Rendering Pipeline

The main focus in the field of computer graphics had been solving the problem of 3D rendering. There are two main schools of thought for 3D rendering: ray tracing-based and rasterization-based rendering, but consumer graphics APIs

and hardware accelerator focuses primarily on rasterization-based technique. We have chosen to adopt a rasterization-based instead of a ray tracing-based 4D rendering pipeline and API in order to best leverage consumer graphics accelerators that are widely available now.

The traditional rasterization-based 3D rendering pipeline can be modified and extended for rendering 4D objects. The 4D pipeline we are adopting in our work is based on previous work by Hanson and Heng.

The primitive used in 4D rendering pipeline is tetrahedron. To render a 3D surface in $\mathbb{E}^4$, we first need to discretize the surface to a hexahedral mesh, then further decompose the surface into a soup of tetrahedra. This process is similar to decomposing a 2D surface in $\mathbb{E}^3$ into a soup of triangles in 3D rendering. Once we get the tetrahedral mesh of the 4D object ready we can feed them into the rendering pipeline. Inside the GPU these tetrahedra will be transformed by 4D model, view and projection matrices and subsequently rasterized into fragments. The fragments will then be shaded by the 4D extension of the Phong shading equation using 4D light sources and written to a 3D frame-buffer. The occluded fragments along the w-coordinate, *i.e.* fourth dimension, will be filtered by a 3D depth buffer if the 4D object we are rendering is opaque. On the other hand proper depth sorting and composition is required if the 4D object is transparent. A final volume rendering step is required to present the 3D frame-buffer on an ordinary 2D display.

Chapter 3

Related Work

We build GL4D upon previous work on volume rendering and 4D visualization. In this chapter, we review work that we base upon and draw inspiration from when working on GL4D.

3.1 General Purpose Processing on Graphics Processing Units

General Purpose Processing on Graphics Processing Units (GPGPU) had been a hot research area in recent years. The thrust of the research is fueled by the possibility of harnessing the large number of parallel floating point computation units in modern GPUs to perform highly parallel computation. Currently there are two categories of APIs to leverage GPU hardware: an older shader-based rendering pipeline and a newer kernel-based computation model. The shader-based rendering pipeline is an evolution of traditional fixed function pipeline by making some stages in the fixed pipeline programmable, examples include OpenGL GLSL, DirectX HL and NVIDIA Cg. On the other hand the kernel-based computation model is a completely new API that allows programs running on CPU to submit kernel programs to GPU for parallel computation, examples are CUDA and OpenCL. These two APIs serve different purposes: shader-based APIs are more suitable when the output is an rendered image

but kernel-based computation APIs are more suitable for computation oriented work. We choose to focus on shader-based APIs in this thesis since our work is primarily rendering-based and would benefit most from a rendering oriented API.

Section 3.2.4 contains a review of related work on applying GPGPU to accelerate volume rendering algorithms.

3.2 Volume Rendering

Volume rendering is a branch of Computer Graphics that concerns with solving the problem of visualizing volumetric data. Volumetric data are usually obtained from sampling taken from real life objects or simulation studies. It is extremely difficult, if not impossible, for human to comprehend raw volume data due to their shear size, visualization of volume data, therefore, is necessary to allow human to fully utilize and extract information hidden behind the raw data.

There are two categories of volumetric data - structured and unstructured. Structured volumetric data have data points defined on a regular grid. On the other hand unstructured volumetric data have data points defined on irregular grid. The problem domain and data collection procedure dictate the format of the volumetric data: experimental procedures such as medical imaging take samples at regular interval in three dimensional space nautrally produce structured volume data and numerical techniques such as finite element method produce unstructured volume data.

Two categories of rendering strategy are available for visualizing volumetric data: indirect volume rendering and direct volume rendering [4]. In indirect volume rendering, the volume data is first converted to an implicit surface and the implicit surface is then rendered in lieu of the original volume data. In

direct volume rendering, as its name implies, renders the volumetric data directly without an implicit surface acting as a proxy. Indirect volume rendering produces a succinct representation of the volume data using implicit surface, while direct volume rendering produces a more holistic view of the volume data.

3.2.1 Indirect Volume Rendering

In indirect volume rendering a constant c is first chosen to generate the implicit surface from the volume data input. To generate the implicit surface each data point x in the volume data are first classified as 'inside' ($x < c$) or 'outside' ($x \geq c$) the implicit surface. An algorithm will then be used to approximate the implicit surface by generating a mesh along the boundary between the 'inside' and 'outside' data points. The classical algorithm for generating the implicit surface is the marching cube algorithm [5]. The marching cube algorithm reads the voxels within a structured volume data one-by-one and a patch of surface is generated from a case table. The summation of these surface patches provides the final geometric model for rendering. A variation of marching cube called marching tetrahedra had been devised to circumvent the patent around the marching cube algorithm and to enable indirect volume rendering on unstructured volume data.

3.2.2 Direct Volume Rendering on Structured Grid

In direct volume rendering (DVR) the voxels in volume data are projected onto the 2D viewing plane directly. The overall architecture for DVR system had not changed much since Levoy's pioneering work [6]. Levoy's proposed pipeline for volume rendering is depicted in Figure 3.1. There are three main steps in the pipeline after data preparation. The first two steps are shading and classification, and the third step is compositing the color and opacity value

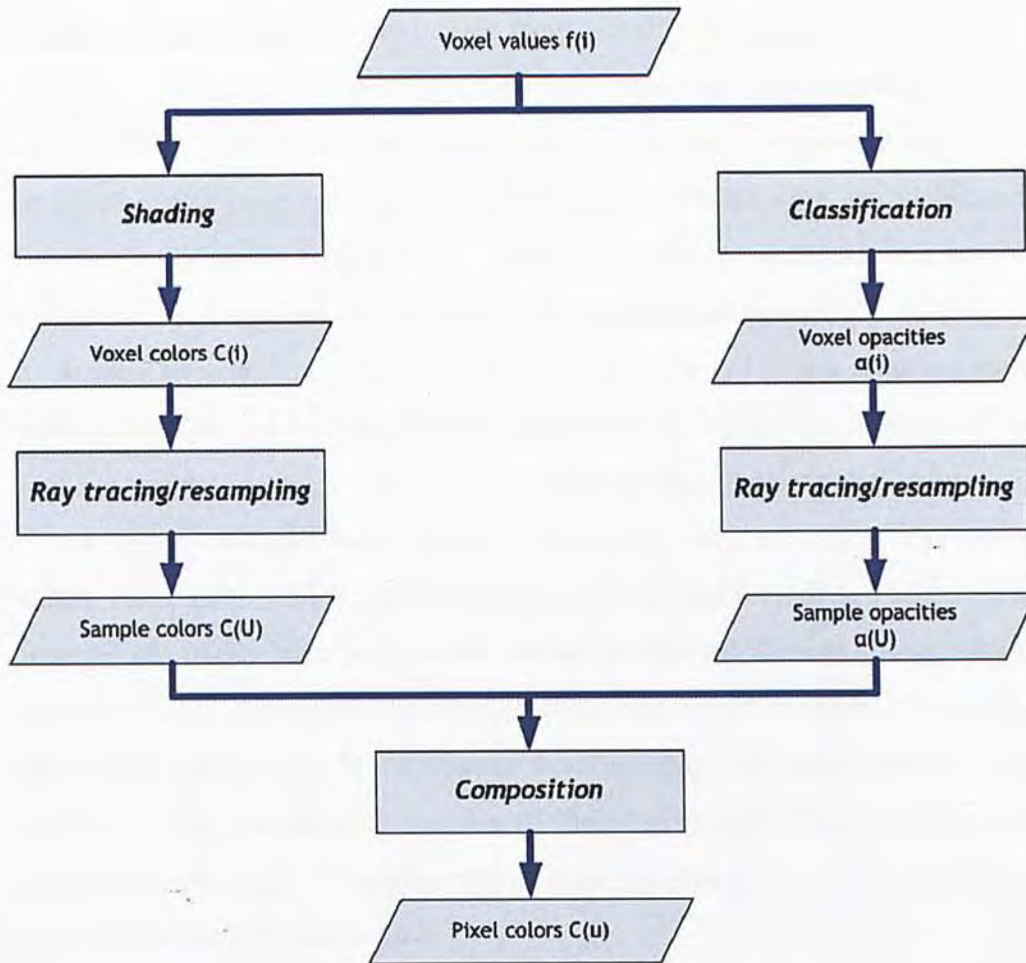

Figure 3.1: The volume rendering pipeline proposed by Levoy [6].

for each voxel to pixels in the 2D viewing plane.

Major Components in DVR

There are five major components in DVR, they are color and opacity transfer functions, gradient estimation, shading and composition. They will be introduced in the following subsections. Section contains a summary of notations used in this section,

Transfer functions are mappings from voxel values $f(\vec{p}) : \mathbb{R}^3 \to \mathbb{R}^n$ to scalar ($n = 1$) or vector ($n > 1$) quantities. There are two common transfer functions for volume data: color and opacity. Transfer functions is important for volume rendering because they control how volume data are visualized. Transfer function by itself is a hot research area in volume rendering as it is crucial to the usefulness and quality of the visualization result [7].

A color transfer function $C(x) : \mathbb{R}^n \to \{x : 0 \leq x \leq 1\}^3$ is a mapping from voxel value $f(\vec{p})$ to a 3-tuple representing a color in the RGB color space. This function is used to assign color to the otherwise meaningless voxel values.

An opacity transfer function $\alpha(x) : \mathbb{R}^n \to \{x : 0 \leq x \leq 1\}$ is a mapping from a voxel value $f(\vec{p})$ to a real number between and including 0 and 1. The presence of opacity transfer function allows translucent display of overlapping isosurfaces and to suppress the display of 'unwanted' voxel values by mapping these voxel values to 0. If the opacity function maps all voxel values to only 0 and 1, in this special case the opacity function is equivalent to binary isosurfaces classification. Therefore the opacity transfer function can be seen as a generalization of classification.

Gradient estimation is an important component both in Levoy's work and other volume rendering systems (*e.g.*, [6, 8, 9, 10, 11]).

The most popular gradient estimation technique is the central difference method and it is also one of the simplest estimation method. Forward and backward differences are used in edge cases.

$$\nabla f(\vec{p}) = \begin{bmatrix} \frac{1}{2}f(\vec{p}+[-1,0,0]) - \frac{1}{2}f(\vec{p}+[1,0,0]) \\ \frac{1}{2}f(\vec{p}+[0,-1,0]) - \frac{1}{2}f(\vec{p}+[0,1,0]) \\ \frac{1}{2}f(\vec{p}+[0,0,-1]) - \frac{1}{2}f(\vec{p}+[0,0,1]) \end{bmatrix}$$

Rheingans and Ebert [9] had studied various gradient method and they found that although there are no systematic difference between the results

generated from different gradient estimation method although the results do differ.

In Levoy's work, the estimated gradients are used to modulate the opacity value in both the isovalue contour surfaces and the region boundary surfaces algorithms. The essence of modulating opacity value by the estimated gradient is to enhance the voxels at the boundary of different voxel values (*i.e.* with large gradient) such as the boundary between organ and fat, and suppress the voxels within a region of uniform voxel values (*i.e.* with low gradient) such as the interior of an organ.

Shading and shadowing of volumetric data is similar to that of geometric shapes. The Phong's shading equation is commonly used [12].

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda} \left(k_d O_{d\lambda} (\vec{N} \cdot \vec{L}) + k_s (\vec{R} \cdot \vec{V})^n \right)$$

The result of the Phong's shading equation I_λ is the intensity of light for a voxel at wavelength λ, $I_{a\lambda}$ is the intensity of ambient light source at wavelength λ, and $I_{p\lambda}$ is the intensity of the directional light source at wavelength λ. $O_{d\lambda}$ is the diffuse color of the voxel at wavelength λ. The constants k_a, k_d and k_s are ambient, diffuse and specular coefficient , and the vectors $\vec{N}$, $\vec{L}$, $\vec{R}$, and $\vec{V}$ are the normal vector, light vector, reflection vector and view vector respectively. Finally, f_{att} is the atmospheric attenuation for the light source.

In the simplest case when $f_{att} = 1$, the intensity of the light source will not be attenuated by the intervening voxels between the voxel being shaded and the light, and shadowing will not occur. Shadowing can be achieved by using a non-constant f_{att} in the shading equation. One example of f_{att} which permits shadowing is $f_{att} = e^{-\int_s^T \mu(t)dt}$ [13]. In the equation, s, T, and $\mu(t)$ are the voxel being shaded, the light source, and the mass density value respectively. The mass density value is determined from a transfer function.

Image composition is the final step of volume rendering for producing a rasterized 2D viewing plane on the frame-buffer with the final opacity $\alpha'(\vec{p})$ and color $C'(\vec{p})$ values assigned to all voxels. In the simplest case $\alpha'(\vec{p}) = \alpha(f(\vec{p}))$ and $C'(\vec{p}) = C(f(\vec{p}))$ where the results of the transfer functions are used directly as the final color and opacity values of the voxel.

There are three major categories of image composition techniques, image-order, object-order, hybrid and domain techniques [13]. Image-order techniques start from the 2D viewing plane and calculate the color and opacity pixel-by-pixel, one widely used image-based technique is ray casting, which was proposed by Levoy [6]. Object-order techniques, on the other hand, work from mesh-to-render to the 2D viewing plane by first sorting the cells in the volume and project each occupied cell to the 2D viewing plane one at a time. Hybrid method tries to combine the advantages of image- and object-order technique, one attempt is the shear warp method developed by Lacroute and Levoy [14]. Finally domain based composition method tries to transform the volume data from spatial domain to another domains such as frequency, and the volume data is then rendered directly from or aided by the transformed voxel data.

Optical model is fundamental to image-based techniques such as ray casting as it describes how light is accumulated and attenuated when passing through the volume. The low-albedo optical model is the simplest optical model where light rays entering the volume are assumed to scattered only once. The low-albedo optical model can be described by the following integral and can be simplified using the following steps [13].

$$I_\lambda(r) = \int_0^{length(r)} C_\lambda(s)\mu(s)e^{(-\int_0^s \mu(t)dt)}ds$$

The result of the equation $I_\lambda(r)$ is the intensity of the ray r at wavelength λ. $C_\lambda(s)$ is the intensity of the color at wavelength λ of voxel s. Similar to the

volumetric shadowing equation, $\mu(s)$ is the mass density value of voxels and $e^{(-\int_0^s \mu(t)dt)}ds$ is the attenuation of light way from the starting point of the ray to the voxel being considered by the outer integral.

The discrete Riemann sum approximation of the above integral is

$$I_\lambda(r) = \sum_{i=0}^{L/\Delta s-1} \left(C_\lambda(i\Delta s)\mu(i\Delta s)\Delta s \prod_{j=0}^{i-1} e^{-\mu(j\Delta s)\Delta s} \right).$$

By assuming that $\alpha(i\Delta s) = 1 - e^{-\mu(i\Delta s)\Delta s}$, the above approximation can be further simplified as

$$I_\lambda(r) = \sum_{i=0}^{L/\Delta s-1} \left(C_\lambda(i\Delta s)\mu(i\Delta s)\Delta s \prod_{j=0}^{i-1} (1 - \alpha(j\Delta s)) \right)$$

Using Taylor series expansion $e^{-\mu(j\Delta s)\Delta s} \approx 1 - \mu(i\Delta s)\Delta s$, therefore $\alpha(i\Delta s) \approx \mu(j\Delta s)\Delta s$. Substituting the result into the approximation results in

$$I_\lambda(r) \approx \sum_{i=0}^{L/\Delta s-1} \left(C_\lambda(i\Delta s)\alpha(i\Delta s) \prod_{j=0}^{i-1} (1 - \alpha(j\Delta s)) \right).$$

The above closed form formula can be written as the following recursive definition and these are the front-to-back composition formulae used by most ray casting algorithms.

$$\begin{aligned} c(0) &= 0 \\ c(i+1) &= C(i\Delta s)\alpha(i\Delta s)\,(1 - \alpha(i)) + c(i) \end{aligned}$$

$$\begin{aligned} \alpha(0) &= 0 \\ \alpha(i+1) &= \alpha(i\Delta s)\,(1 - \alpha(i)) + \alpha(i) \end{aligned}$$

3.2.3 Direct Volume Rendering on Unstructured Grid

Direct volume rendering on unstructured grid requires specialized algorithms or adaptation of algorithm that are applicable to structured grid. Some of the popular algorithms includes ray casting [15], projected tetrahedra [16, 17], scan plane [18] and slicing [19].

Two popular tetrahedral mesh rendering algorithms are projected tetrahedra (PT) algorithm proposed by Shirley and Tuchman and rasterization by view axis aligned slicing plane. The projected tetrahedra algorithm works by transforming the tetrahedra mesh and create two to four semi-transparent triangular proxy primitives for rendering the original tetrahedron. Slicing-based volume rendering algorithm, on the other hand, renders the tetrahedral mesh by slicing the tetrahedral mesh by slicing plane at regular interval, and the images formed by successive slicing planes are composited to form the final image.

3.2.4 Acceleration of DVR

A lot of research effort had been put into accelerating DVR in various ways. These acceleration can be classified into three main categories: algorithmic improvements, texture mapping and parallel computation using GPGPU.

Algorithmic improvements

DVR algorithms can be improved algorithmically. Starting from adaptive ray termination and hierarchical spatial enumeration proposed by Levoy [20], a lot of research effort had been put into improving both the speed and reduce the memory usage of the volume renderer. This research direction is driven by the development of data acquisition hardware in clinical procedures which are capable to produce data at a much higher resolution than the data set ($\geq 512^3$) usually used in research projects [10].

GPU accelerated volume rendering

GPU accelerated volume rendering provides an alternative means to speed up DVR by using modern graphics hardware. There are two approaches to accelerate volume rendering using GPU: image-based algorithms such as 3D texture mapping and ray casting, and object-based algorithms like projected tetrahedra.

Hardware support of 3D texture mapping had been used by various researchers in volume rendering system because of the fast trilinear interpolation implementation provided by GPU. One of the earliest application of 3D texture mapping [21] uses the final color $C'(\vec{p})$ and opacity value $\alpha'(\vec{p})$ of each voxel as the RGBA values of the 3D texture, and the 3D texture is then mapped onto a series of 2D planes as shown in Figure 3.2. Only the composition stage in the volume rendering pipeline is moved to the graphics hardware in this application. Later applications of 3D texture mapping tries to offload more pipeline stages from CPU to the graphics hardware. Although using 3D texture mapping can achieve higher frame-rate, rendering quality is sacrificed due to the restriction of texture memory in using single precision floating point format (32-bit) on common graphics cards [22].

Another image-based GPU accelerated volume rendering technique is ray casting. Krüger and Westermann [23] had proposed an implementation of ray casting algorithm with optimizations that can run on GPU entirely. An important difference between using 3D texture mapping and GPU-based ray casting is that optimizations such as adaptive termination and empty space skipping can be implemented in GPU-based ray casting to save fragment (pixel) operations on transparent voxels. GPU-based ray casting algorithm has two stages: the first stage determines the entry and exit points of each ray from each pixel in the 2D viewing plane, this is accomplished by rendering the volume bounding box (Figure 3.3); the second stage involves a multi-pass rendering that

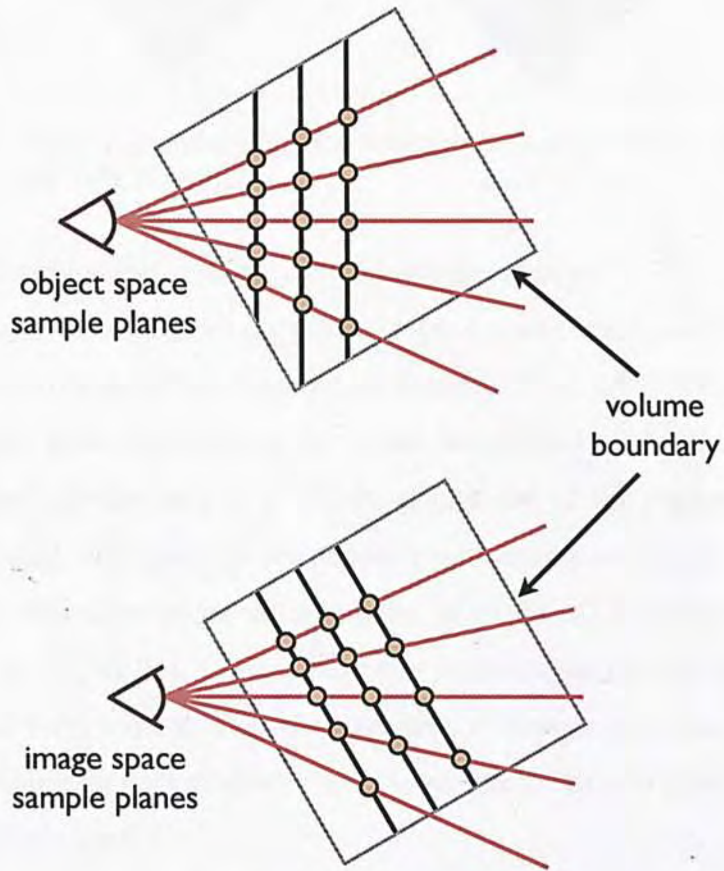

Figure 3.2: 3D texture mapping technique proposed by Cullip and Neumann [21].

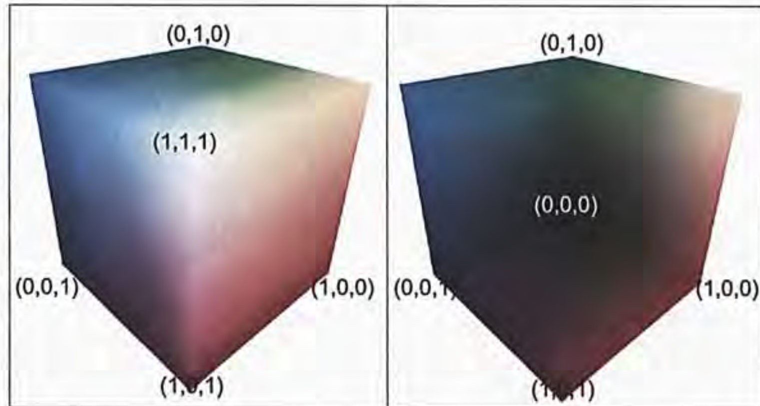

Figure 3.3: Volume bounding box for determining entry and exit point of rays casted from viewing plane [23].

calculate the color and opacity accumulated for each ray.

The projected tetrahedra algorithm [16] is a prime example of object-based volume rendering technique that can be accelerated by GPU. Projected tetrahedra begins with decomposing the input hexahedral mesh into tetrahedral mesh. These tetrahedra will be classified into one of the classes outlined in Figure 3.4 and subsequently decomposed into triangular strips (Figure 3.5. Finally the triangular strips can be rendering to the 2D frame-buffer.

Wylie et al. [24] had implemented the projected tetrahedra algorithm on GPU. They have implemented the tetrahedron classification and decomposition algorithms on vertex shader and made projected tetrahedra algorithm running entirely on GPU.

3.3 4D Visualization

Early research on visualization of high-dimensional geometry includes the work by Noll [25] and Banchoff [26, 27], who exploited 3D computer graphics methods to display 4D objects. Methods exploited in a variety of early work [28, 29, 30, 31, 32, 33] included wireframe representations, hyperplane slicing, color coding, view transformations, projection, and animation.

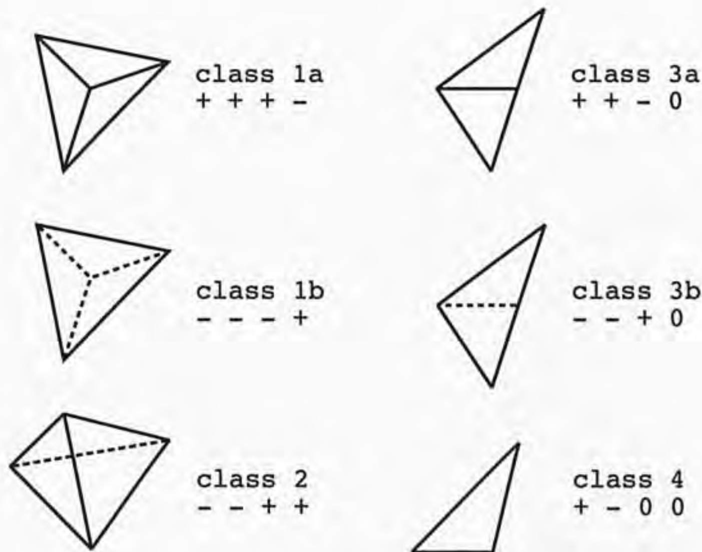

Figure 3.4: Tetrahedra classification [16].

Burton et al. [34, 35] and Hanson and Heng [36, 37, 38] proposed various frameworks that included lighting models for the visualization of 4D geometries and extended the methods of 3D rendering to the fourth dimension. Rendering 3D objects onto a 2D screen was replaced by projecting 4D geometry into a 3D frame-buffer volume, and 4D depth buffer to cull occluded fragments in the 4D-to-3D projection. Hanson and Heng also proposed a thickening mechanism to support converting 2-manifolds immersed in 4 dimensional Euclidean space $\mathbb{E}^4$ to renderable 3-manifolds. The resulting volume frame-buffer calls for 3D volume rendering methods to expose the internal structure of the projected 4D geometry. Transparent 4D objects rendering and, hence, alpha blending along the 4D projection direction had not been studied in previous work. An alternative volume rendering to expose geometric structure after 4D-to-3D projection was suggested by Banks [39] , who employed principal curves on surfaces, transparency, and screen-door effects to highlight intersections in the projected geometry; in addition, Banks [40] proposed a general mechanism to compute diffuse and specular reflection of a k-manifold embedded in n-space.

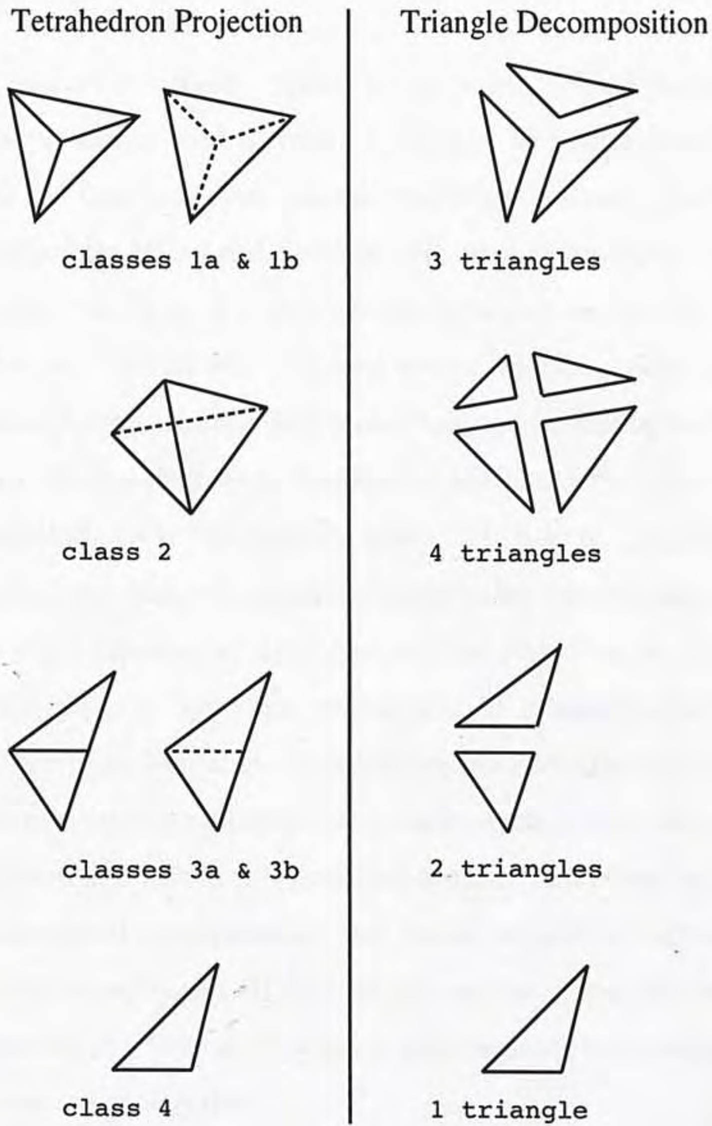

Figure 3.5: Tetrahedra decomposition [16].

Hanson and Cross [41, 42] developed techniques implementing 4D rendering with the Shirley-Tuchman volume method [16]. assuming that the objects in 4D are static and occlusion-free in the 3D frame-buffer. Such methods cannot provide real-time occlusion computation and have limited interactivity compared to our approach.

Previous researches closely related to our work include Feiner and Beshers' [43] 'worlds within worlds' interface system to manipulate and explore high-dimensional data space via nested coordinate systems. Another related system developed by Miller and Gavosto [44]; used sampling methods to render and visualize 4D slices of n-dimensional data such as fractals and satellite orbits. Duffin and Barrett [45] proposed a user interface design to carry out n-dimensional rotation. Hanson [46] generalized the 3D rolling ball control [47] to manipulate the six degrees of freedom of 4D rotations. Among other interesting contributions to the field are those of Egli et al. [48], who proposed a moving coordinate frame mechanism to generalize the sweeping method for representing high-dimensional data, the work of Bhaniramka et al. [49], who explored isosurfacing in high-dimensional data by a marching-cube-like algorithm for hypercubes, and that of Neophytou and Mueller [50], who investigated the use of splatting to display 4D datasets such as time-varying 3D data. Recently, Hanson and Zhang [51] proposed a multimodal user interface design that integrates visual representation and haptic interaction, allowing users to simultaneously see and touch 4D objects; this approach was then extended [52] to exploit the idea of a reduced-dimension shadow space to directly manipulate higher-dimensional geometries.

Chapter 4

GL4D: Hardware Accelerated Interactive 4D Visualization

GL4D is a platform for hardware accelerated 4D visualization. The core of GL4D is a GPU-friendly implementation of a 4D rendering pipeline designed to be accelerated by modern graphics processor. GL4D is able to visualize 4D objects at interactive speed with high quality rendering. While GL4D is based on previous work on 4D visualization and volume rendering, it is not a simple and trivial translation of the basic 4D rendering pipeline to GPU, a combination of algorithmic simplification and implementation optimizations had been employed to build the 4D rendering engine on top of a GPU originally designed for rendering 3D surface models.

GL4D consists of two major components: a preprocessing component for generating tetrahedral mesh from parametric equations on CPU and a core rendering pipeline for rendering tetrahedral mesh on GPU. Both the preprocessing component and the core rendering pipeline will be discussed in details in the following sections. Figure 4.1 provides a schematic overview of GL4D.

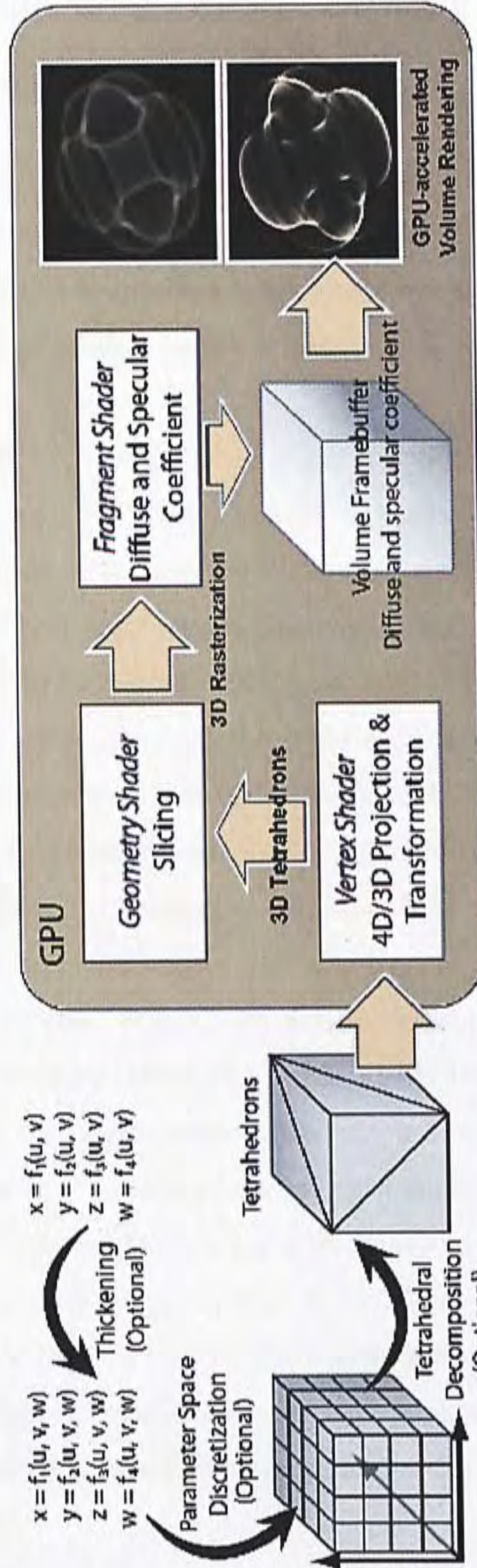

Figure 4.1: An overview of GL4D.

4.1 Preprocessing: From Equations to Tetrahedral Mesh

Akin to a traditional 3D rendering pipeline that uses triangles (2-simplex) as primitive, the GL4D rendering pipeline uses tetrahedra (3-simplex) as primitive. Therefore 3-manifolds specified in parametric equations have to be discretized to tetrahedral meshes before it can be rendered by the rendering pipeline.

A 3-manifold $\mathcal{M}^{3\to4}(t,u,v) \in \{\mathbb{R}^3 \to \mathbb{E}^4\}$ is specified by a set of parametric equations $\mathcal{M}^{3\to4}_x(t,u,v)$, $\mathcal{M}^{3\to4}_y(t,u,v)$, $\mathcal{M}^{3\to4}_z(t,u,v)$ and $\mathcal{M}^{3\to4}_w(t,u,v)$. The generation of the tetrahedral mesh for 3-manifold requires two sets of parametric equation: the positional parametric equations given above and the set of parametric equations for normals $\mathcal{M}^{3\to4}_{normal}(t,u,v) \in \{\mathbb{R}^3 \to \mathbb{E}^4\}$.

These two sets of parametric equations are sampled at regular interval in the parametric space to create a hexahedral mesh, and the hexahedral mesh are further decomposed into a tetrahedral mesh via 5- or 6-tetrahedral decomposition algorithm (Figure 4.2). The major differences between the two algorithms are the number of tetrahedra output and whether the decomposition line of the opposite face matches. While 5-tetrahedra decomposition produces less tetrahedra output, adjacent hexahedra needs to be decomposed in opposite orientation to make the decomposition line matches for adjacent hexahedra. This limits the number of hexahedra decomposed along each parameter to be even-numbered if the 3-manifold is closed to make the decomposition line of the first and last hexadedra match. Both 5- and 6-tetrahedra decomposition are supported by GL4D but 5-tetrahedra decomposition is used by default to keep the size of tetrahedra mesh small whenever possible.

Section 5.1 gives the parametric equations for the surfaces that can be rendered with GL4D.

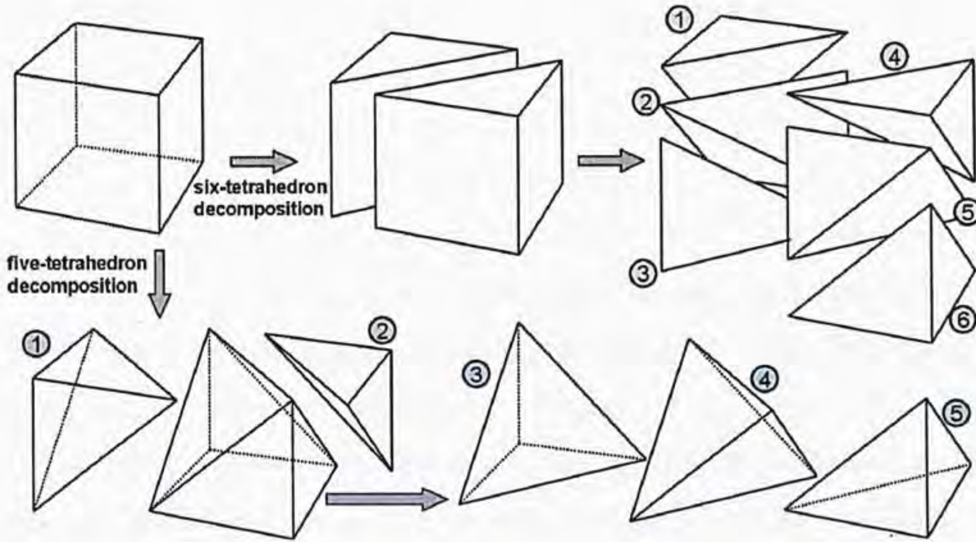

Figure 4.2: Two possible ways of decomposing a hexahedral cell into tetrahedra: 6-tetrahedra (top) and 5-tetrahedra (bottom).

4.2 Core Rendering Pipeline: OpenGL for 4D Rendering

While OpenGL is primarily an API for rendering 3D graphics, most of its internal data path handles 4-vectors and 4×4 matrices—with the notable exception of normal vectors—due to its use of homogeneous coordinate system for translation and perspective projection in 3D space. Recent development in OpenGL introduces programmable shaders that allow customization and augmentation to the otherwise hardwired OpenGL rendering pipeline. The combination of 4-vector data path and programmable shaaders in OpenGL opens the door for GL4D to utilize and extend OpenGL for 4D rendering.

GL4D utilizes vertex, geometry and fragment shaders to implement the 4D rendering pipeline. The vertex shader applies transformation and perspective calculation to the 4D vertices of tetrahedra and suppresses perspective division in fixed function pipeline, the geometry shader [53] slices the tetrahedra

to rasterize the tetrahedra into voxels and the fragment shader computes the Phong shading equation from 4D light sources to shade the voxels. One complete rendering pass through all shaders, vertex, geometry, and fragment, are needed to render one slice of volume frame-buffer.

The GL4D core rendering pipeline implemented in GL4D is separated into several stages: vertex data upload, slice-based multi-pass tetrahedral mesh rendering and a back-to-front composition to form the final image. The following sections will introduce and describe these stages.

In GL4D the process of rendering to a 3D frame-buffer is implemented by multi-pass rendering, with a different *slicing plane* defined for each pass. A slicing plane is an axis-aligned plane along the z-axis after projection from 4D to 3D and corresponds to a slice of voxels within the 3D frame-buffer.

4.2.1 Vertex Data Upload

The tetrahedral mesh for the 3-manifold generated in the preprocessing step is uploaded entirely to the GPU memory to eliminate the need to upload vertex data for each rendering pass. The vertex data are stored in vertex buffer object (VBO) to allow more efficient storage using `GL_ELEMENT_ARRAY_BUFFER`. `GL_ELEMENT_ARRAY_BUFFER` adds an additional level of indirection when drawing primitives from VBO: `GL_ELEMENT_ARRAY_BUFFER` stores indices which indirectly refers to vertex data stored in other vertex arrays. This level of indirection allows vertex data to be shared among multiple primitives.

4.2.2 Slice-based Multi-pass Tetrahedral Mesh Rendering

GL4D features a slice-based algorithm for tetrahedral mesh rendering: n rendering passes are needed to render n slices within the 3D frame-buffer volume

and the complete OpenGL vertex-geometry-fragment shader pipeline is invoked for each pass. The division of work among the shaders will be described in the coming sections.

Choice of Volume Rendering Algorithm for Tetrahedral Mesh

While the basic 4D rendering pipeline leverages a 3D frame-buffer [37] for efficient texture-based volume rendering , OpenGL supports rendering to a 3D texture a-slice-at-a-time only. This limitation is the biggest challenge in implementing the core rendering pipeline in OpenGL and we have considered two strategies for solving this problem.

The first strategy is to treat the tetrahedral mesh after projection to 3D view space as unstructured grid and perform direct rendering on the projected unstructured tetrahedral grid directly using algorithm such as Shirley and Tuchman's Projected Tetrahedra [16]. The biggest advantage of this strategy is low algorithmic complexity as the 3D frame-buffer is eliminated and the projected tetrahedral mesh is volume rendered directly onto the 2D screen. While rendering unstructured tetrahedral grid to screen in one pass is a very attractive solution, Projected Tetrahedra assumes the whole tetrahedron contributes to the final image without any occlusion. While this assumption is valid for volume rendering standard 3D tetrahedral mesh dataset, GL4D requires a 3D depth buffer in place to perform depth buffering in 4D space properly. Difficulties in generating the 3D depth buffer and maintaining the whole depth buffer in graphics memory make the choice of rendering the projected tetrahedral mesh directly onto the screen in one single pass an implausible choice for GL4D.

The second strategy is to use slice-based algorithm to volume render the tetrahedral mesh. While slice-based algorithm requires multiple rendering passes to generate one frame, incurring high performance penalty, the slice-based nature of the algorithm allows us to maintain the depth buffer for the

current slice only and, more importantly, makes it possible to utilize hardware-based z-buffer for efficient 4D depth testing.

Therefore the GL4D core rendering pipeline is evolved from slice-based rendering algorithm that, we believe, strikes a good balance between rendering speed and memory consumption.

Vertex Shader

The vertex shader is responsible for transforming the input 4D vertices to normalized device space within the 3D frame-buffer and 4D normals to 4D eye space. The transformed vertex coordinates in 3D normalized device space, the transformed normal in 4D eye space, and the w-coordinate of the vertex in eye space are attached to the vertex output and sent to subsequent shaders for the purpose of rasterization, 4D lighting calculation, and occluded fragment removal respectively.

The calculation in vertex shader includes 4D model view transformation and 4D perspective projection. Since GL4D have exhausted all four components of the 3D homogeneous coordinates to specify a true 4D coordinate space, GL4D needs to perform explicit perspective calculation instead of using perspective division from homogeneous coordinate system. Two sets of transformations transform the 4D vertex input: the first transformation is for manipulating the object in 4D space and the second transformation is for rotating the 3D frame-buffer volume.

The following equations describe the first set of transformation with vertex $\vec{p}_{\text{In4D}}$, normal $\vec{n}_{\text{In4D}}$ and model view matrix $\mathbf{M}_{\text{ModelView4D}}$.

$$\vec{p}_{\text{Out4D}} = \mathbf{M}_{\text{ModelView4D}}\vec{p}_{\text{In4D}}$$

$$\vec{n}_{\text{Out4D}} = \mathbf{M}^{-1}_{\text{ModelView4D}}\vec{n}_{\text{In4D}}$$

The above equation can be extended to incorporate 4D perspective transformation with an axis-aligned near plane at $w = \text{near}_w$ and vanishing point

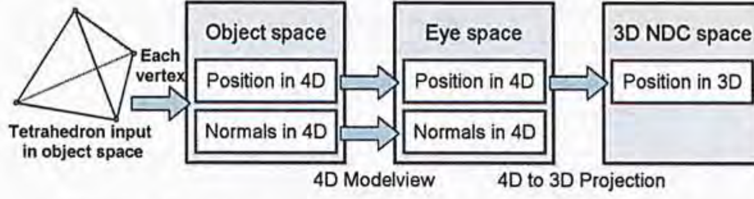

Figure 4.3: 4D transformations in vertex shader.

at $(0, 0, 0, \text{vanishing}_w)$.

$$\vec{p}_{\text{Out4D}} = \frac{\text{vanishing}_w - \text{near}_w}{\text{vanishing}_w - (\mathbf{M}_{\text{ModelView4D}}\vec{p})_w}(\mathbf{M}_{\text{ModelView4D}}\vec{p})$$

The second set of transformation follows after projection of the 4D vector $\vec{p}_{\text{Out4D}}$ to 3D by discarding the w-coordinate and becomes $\vec{p}_{\text{In3D}}$. The model view projection matrix $\mathbf{M}_{\text{ModelViewProj3D}}$ is used to transform the projected vector $\vec{p}_{\text{In3D}}$.

$$\vec{p}_{\text{In3D}} = \text{proj}_{xyz}(\vec{p}_{\text{Out4D}})$$
$$\vec{p}_{\text{Out3D}} = \frac{\mathbf{M}_{\text{ModelViewProj3D}}\vec{p}_{\text{In3D}}}{(\mathbf{M}_{\text{ModelViewProj3D}}\vec{p}_{\text{In3D}})_w}$$

$\vec{p}_{\text{Out3D}}$ is finally translated by amount $(0, 0, -z_{slice})$ for slicing plane $z = z_{slice}$ and the original depth value $\vec{p}_{\text{Out4D}w}$ is piggy-backed to the w-coordinate of the output vector $\vec{p}_{\text{Out}}$ to support hardware 4D depth testing.

$$\vec{p}_{\text{Out}} = \left((\vec{p}_{\text{Out3D}} - (0, 0, z_{slice}))_{xyz}, \vec{p}_{\text{Out4D}w}\right)$$

GL4D does not currently support translation in 4D, but it would be trivial to achieve this using an extra shader variable for the 4D translation vector. Figure 4.3 summarizes the transformation calculations done by vertex shader in GL4D.

Geometry Shader

The need of depth testing the fragments in GL4D ruled out the possibility to reuse existing algorithms for tetrahedra rasterization. Therefore we have devised a novel geometry shader-based tetrahedra slicing algorithm that supports efficiently depth test each fragment for tetrahedra in tetrahedral mesh.

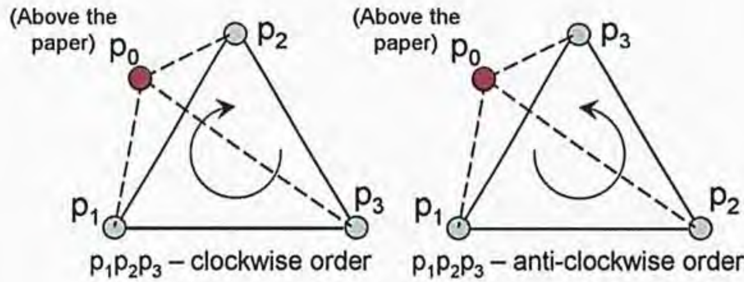

Figure 4.4: Two possible vertex ordering in a tetrahedron. p_0 is above the paper and p_1, p_2 and p_3 are on the paper.

The geometry shader [53] receives transformed vertices of one tetrahedron from vertex shader, performs back-tetrahedra culling, and calculates the intersection between the input tetrahedron and the axis-aligned plane at the specified w-coordinate. There are three types of intersection between a plane and a tetrahedron: no intersections, a triangle, or a quadrilateral. The input to the geometry shader is one tetrahedron specified by a 4-vertex geometry-shader-specific primitive `GL_LINES_ADJACENCY_EXT`. The output of geometry shader is one triangle strips specifying the intersection between the input tetrahedron and the slicing plane. The collective sum of these triangle strip outputs forms the cross section between the entire tetrahedral mesh and the slicing plane.

Back-tetrahedra culling is used in GL4D to eliminate back-facing or degenerated tetrahedra from the rendering pipeline and avoid them from participating in subsequent pipeline stages as they do not contribute to the rendered image. Similar to back-face culling in traditional 3D rendering, back-facing culling in GL4D requires the vertices of tetrahedra in the tetrahedral mesh to be specified in a consistent order. The four vertices (p_0, p_1, p_2, p_3) of a tetrahedron can be specified either in clockwise or anti-clockwise order, depending on the spatial ordering of p_1, p_2 and p_3 when observed from p_0. Figure 4.4 illustrates the two possible (clockwise and anti-clockwise) vertex orderings when specifying a tetrahedron.

GL4D distinguishes front- and back-facing tetrahedra by calculating the face normal of a tetrahedron:

$$\text{Face normal of a tetrahedron} = \begin{vmatrix} v_{1x} & v_{2x} & v_{3x} & \hat{x} \\ v_{1y} & v_{2y} & v_{3y} & \hat{y} \\ v_{1z} & v_{2z} & v_{3z} & \hat{z} \\ v_{1w} & v_{2w} & v_{3w} & \hat{w} \end{vmatrix}$$

where $\vec{v}_i = (v_{ix}, v_{iy}, v_{iz}, v_{iw}) = p_i - p_0$, with $i = 1$, 2, and 3, are the three vectors forming the edges of tetrahedron tetrahedron from the vertex p_0. Furthermore, since back-face culling requires only the sign of the w-coordinate in the face normal, we can simplify the computation to:

$$\begin{aligned} w\text{−coordinate of the face normal} &= \begin{vmatrix} v_{1x} & v_{2x} & v_{3x} \\ v_{1y} & v_{2y} & v_{3y} \\ v_{1z} & v_{2z} & v_{3w} \end{vmatrix} \\ &= v_{1xyz} \cdot (v_{2xyz} \times v_{3xyz}). \end{aligned}$$

The final equation is the signed volume of tetrahedron, and GL4D treats tetrahedra with negative signed volume as back-facing. Furthermore, tetrahedra with zero signed volume is degenerated and should also be culled.

Slice-based tetrahedra rasterization rasterizes the tetrahedral mesh slice-by-slice by calculating the intersection between a slicing plane and the tetrahedra (Figure 4.5) in the mesh using Marching Tetrahedron algorithm [54] in geometry shader.

Geometry shader can assume the slicing plane is always located at $z = 0$ as the vertex output $\vec{p}_{\text{Out}}$ from vertex shader is already translated by the amount $(0, 0, -z_{slice})$ for slicing plane $z = z_{slice}$.

The geometry shader first calculates a *sign vector* $\vec{v}_{\text{sign}}$ for the tetrahedron input $(\vec{p}_0, \vec{p}_1, \vec{p}_2, \vec{p}_3)$ by filtering z-coordinate of each vertex through the sign

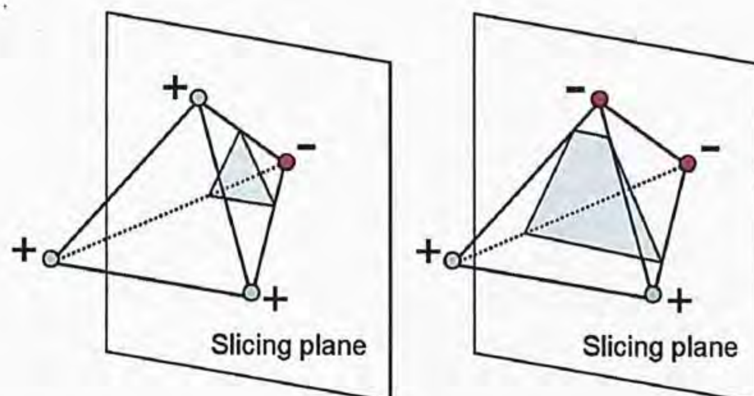

Figure 4.5: Two possible intersection between a tetrahedron and a plane: a triangle or a quadrilateral

operator. The sign operator is defined as $\text{sign}(x) = 0$ if $x \leq 0$ and $\text{sign}(x) = 1$ if $x > 0$.

$$\vec{v}_{\text{sign}} = (sign(\vec{p}_{0z}), sign(\vec{p}_{1z}), sign(\vec{p}_{2z}), sign(\vec{p}_{3z}))$$

There are 2^4 possible sign vectors and it is used to lookup which edges of a tetrahedron intersects with slicing plane $z = 0$ from a lookup table stored in an integer texture (Table 4.1).

Finally the geometry shader outputs 3 or 4 vertices with vectors for position and normal linearly interpolated along the edges specified by the lookup table. The output vertices form a triangle strip (`GL_TRIANGLE_STRIP`) and represents the intersection between the slicing plane $z = 0$ and the tetrahedron.

Hardware Accelerated 4D Depth Testing

GL4D utilizes OpenGL hardware depth buffer to cull occluded fragments along the fourth dimension. Enabling 4D depth testing can significantly improve rendering performance by reducing fragment shader executions.

The z-coordinate of the output vertices from geometry shader is guaranteed to be 0 since these vertices must lie on the slicing plane $z = 0$. This observation allows us to overwrite the z-coordinate of the vertex output by

sign vector (v_{sign})	set of tetrahedron edges intersecting plane $z = z_{slice}$
$- - - -$	$\emptyset$
$- - - +$	$\{(v_0, v_3), (v_1, v_3), (v_2, v_3)\}$
$- - + -$	$\{(v_0, v_2), (v_1, v_2), (v_2, v_2)\}$
$- - + +$	$\{(v_0, v_2), (v_1, v_2), (v_0, v_3), (v_1, v_3)\}$
$- + - -$	$\{(v_0, v_1), (v_2, v_1), (v_3, v_1)\}$
$- + - +$	$\{(v_0, v_1), (v_0, v_3), (v_2, v_1), (v_2, v_3)\}$
$- + + -$	$\{(v_0, v_1), (v_0, v_2), (v_1, v_3), (v_2, v_3)\}$
$- + + +$	$\{(v_0, v_1), (v_0, v_2), (v_0, v_3)\}$
$+ - - -$	$\{(v_0, v_1), (v_0, v_2), (v_0, v_3)\}$
$+ - - +$	$\{(v_0, v_1), (v_0, v_2), (v_1, v_3), (v_2, v_3)\}$
$+ - + -$	$\{(v_0, v_1), (v_0, v_3), (v_2, v_1), (v_2, v_3)\}$
$+ - + +$	$\{(v_0, v_1), (v_2, v_1), (v_3, v_1)\}$
$+ + - -$	$\{(v_0, v_2), (v_1, v_2), (v_0, v_3), (v_1, v_3)\}$
$+ + - +$	$\{(v_0, v_2), (v_1, v_2), (v_2, v_2)\}$
$+ + + -$	$\{(v_0, v_3), (v_1, v_3), (v_2, v_3)\}$
$+ + + +$	$\emptyset$

Table 4.1: Lookup table for acclerating marching tetrahedra

the w-coordinate from the original transformed 4D vertex $\vec{p}_{\text{Out4D}}$ before projection to 3D. This is made possible by having the vertex shader piggy backing $\vec{p}_{\text{Out4D}w}$ to its output $\vec{v}_{\text{Out}}$ and overwriting z-coordinate of the vertex output from geometry shader by $\vec{p}_{\text{Out4D}w}$.

Fragment Shader

Fragment shader in GL4D is responsible for calculating color and opacity values for each voxel in 3D frame-buffer volume. The fragment shader employs per-voxel Phong shading extended to four dimensional light source to obtain maximum rendering quality. The following equation formally presents the Phong's shading equation for calculating the shading value of a fragment at position $\vec{p}$ with normalized normal $\hat{N}$. The ambient constant is i_a, and there are n point light sources with each light characterized by position vector $\vec{l}$ plus diffuse and specular constants i_d and i_s. The constants k_a, k_d, k_s and n_s

are global factor for ambient, diffuse, specular and specular exponent respectively. We further need to compute the unit vectors $\hat{L}_l = \text{normalize}(\vec{l}_l - \vec{p})$ and $\hat{R}_l = \text{normalize}(2(\hat{L}_l \cdot \hat{N})\hat{N} - \hat{L}_l)$ for each light source l. Finally we take $\hat{V} = (0, 0, 0, 1)$ since GL4D assumes a constant viewing vector.

$$I = k_a i_a + \sum_{\text{lights}} \left(k_d i_d \max\left(\hat{L}_l \cdot \hat{N}, 0\right) + k_s i_s \max\left(\left(\hat{R}_l \cdot \hat{V}\right)^{n_s}, 0\right)\right)$$

The above shading equation is good for rendering single-sided surface. The two max operators will be replaced by abs operators when the surface-to-render is two-sided.

4.2.3 Back-to-front Composition

While conceptually we volume render the whole 3D frame-buffer after completing all rendering passes of the multi-slice tetrahedral mesh rasterization algorithm, this 3D frame-buffer does not actually exist in GL4D. In GL4D the 3D frame-buffer are rendered and composited in back-to-front order on-the-fly: the furthest slice is rendered and composited to the output and subsequent slices are blended to the output using ordinary alpha blending equation right after they are rendered. The need to maintain the whole 3D frame-buffer texture in graphics memory is therefore eliminated.

4.3 Advanced Visualization Features in GL4D

Building upon the basic 4D rendering pipeline, GL4D supports several advanced rendering techniques. This section describes these advanced rendering options made possible by GL4D, and how these options enable better appreciation of 3-manifold in 4D space.

4.3.1 Stereoscopic Rendering

Unlike ordinary 3D rendering, the voxels in the 3D frame-buffer does not cast shadow upon itself. This made interpreting the intricate 3-manifold projection in the 3D frame-buffer more difficult than usual as the depth cues such as light-shadow contrast we have accustomed to are unavailable. One way to overcome this limitation is by using stereoscopic rendering to provide 3D information via binocular vision instead of traditional light-shadow depth cues. GL4D supports stereoscopic rendering of the 3D frame-buffer at the expense of executing at half the normal frame-rate.

Stereoscopic rendering in GL4D is accomplished by rendering the 3D frame-buffer twice, one for left eye and the other for right eye, with two slightly different asymmetric frustum projection for projecting the voxels in 3D frame-buffer to 2D screen. The asymmetric frustum algorithm adopted by GL4D is adapted from description by Paul Bourke [55]:

Algorithm 1: `glFrustum` setup for the image for left eye.

```
1 ScreenHalf ← 0.5 ∗ ScreenWidth ∗ FrustumFar / FrustumNear ;
2 glFrustum(
      // Left
3     −(ScreenHalf − Halflod) ∗ FrustumNear / FrustumFar,
      // Right
4     +(ScreenHalf + Halflod) ∗ FrustumNear / FrustumFar,
      // Bottom
5     −(0.5 ∗ ScreenHeight),
      // Top
6     0.5 ∗ ScreenHeight,
7     FrustumNear, FrustumFar
8 );
9 glTranslated(−Halflod, 0, 0);
```

Algorithms 1 and 2 show how `glFrustum` is setup differently for left and right eyes in GL4D to allow stereoscopic rendering. The constants ScreenWidth and ScreenHeight are the size of the screen; FrustumNear and FrustumFar are the z-coordinate of the near and far plane along the z-axis; and Halflod is

Algorithm 2: `glFrustum` setup for the image for right eye.

```
1 ScreenHalf ← 0.5 * ScreenWidth * FrustumFar / FrustumNear ;
2 glFrustum(
      // Left
3     −(ScreenHalf + Halflod) * FrustumNear / FrustumFar,
      // Right
4     +(ScreenHalf − Halflod) * FrustumNear / FrustumFar,
      // Bottom
5     −(0.5 * ScreenHeight),
      // Top
6     0.5 * ScreenHeight,
7     FrustumNear, FrustumFar
8 );
9 glTranslated(+Halflod, 0, 0);
```

half of interocular distance—the greater the interocular distance the more the stereoscopic polarity between the images for left and right eyes.

4.3.2 False Intersection Detection

Similar to a 3D object without self intersections itself can produce 2D projection with self intersections, a 4D object without self intersection can also produce projection with self intersections when projected to 3D frame-buffer. GL4D has the ability to detect these false intersections using the min-max depth buffer technique [56] and display these self intersections to users.

False intersection detection works by rendering each slice in the 3D frame-buffer two times: the first pass produces a min-max depth buffer for the slice $z = z_{slice}$ and the second pass renders the object normally.

A min-max depth buffer records the minimum and maximum depth value of all fragments $\text{frag}_i = (x, y, z_{slice}, w_i)$ arriving at the same pixel (x, y) of the min-max depth texture, *i.e.*

$$\text{MinDepth}_0(x, y) = \min\left(\{w_i : \forall\, \text{frag}_i = (x, y, z_{slice}, w_i)\}\right)$$
$$\text{MaxDepth}_0(x, y) = \max\left(\{w_i : \forall\, \text{frag}_i = (x, y, z_{slice}, w_i)\}\right).$$

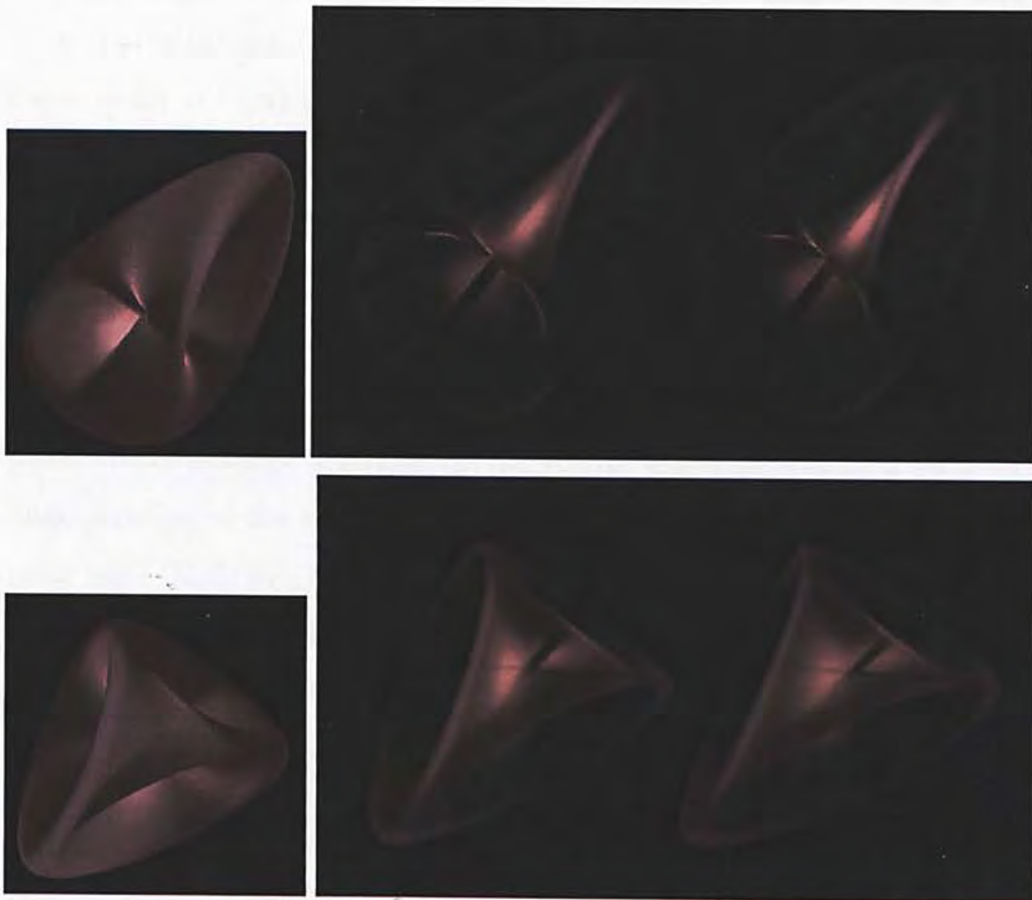

Figure 4.6: Steiner surface in divergent stereoscopic rendering.

This min-max depth buffer is computed using the `GL_MAX` blending equation and writing $(\text{frag}_{i,w}, -\text{frag}_{i,w}, 0, 0)$ as the output color value from fragment shader for fragment frag_i. This process creates a 2D texture with pixels recording the maximum and minimum depth value for fragments arriving at each pixel. The minimum and maximum depth value differs only when more than one fragment with different depth value had accumulated to the same voxel, *i.e.* $\text{MinDepth}_0(x, y) \neq \text{MaxDepth}_0(x, y)$. In such case the voxel in 3D frame-buffer at (x, y) is deemed a false intersection and can be rendered with special color in the second normal rendering pass.

4.3.3 Transparent 4D Objects Rendering

While the volume rendered image of the projected 3-manifold shows some degree of transparency, the 4D object being rendered is actually opaque. The transparency is added artificially in the volume rendering step to allow better understanding of the intricate structure inside the 3D frame-buffer. On the other hand GL4D also supports rendering a transparent 4D object by adapting and extending the dual depth peeling algorithm [56] to 4D rendering.

The basic 4D rendering pipeline assumes the opaque 3-manifolds, therefore 4D depth testing is done along the fourth dimension to eliminate occluded fragments. To support rendering transparent 4D object we first need to disable 4D depth testing and then sort the fragments compositing on the same voxel in back-to-front order in order to composite them correctly. Consider the scenario where orthographic projection is used to project n fragments $\text{frag}_0 = (x, y, z, w_0)$, $\text{frag}_1 = (x, y, z, w_1)$, ..., $\text{frag}_{n-1} = (x, y, z, w_{n-1})$ with $w_0 < w_1 < ... < w_{n-1}$, to the same voxel (x, y, z) in the 3D frame-buffer volume. For opaque 4D object all but the nearest fragment $f_0 = (x, y, z, w_0)$ is discarded and only the nearest fragment can be written to the voxel (x, y, z) in the frame-buffer volume. On the contrary when rendering transparent 4D object

all fragments $\text{frag}_i = (x, y, z, w_i)$ have to be composited to the voxel (x, y, z).

GL4D supports rendering transparent 4D objects by performing 4D depth peeling algorithm on every slice in the 3D frame-buffer volume. To render a slice at $z = z_{slice}$ GL4D first needs to obtain an initial min-max depth buffer. Recall that a min-max depth buffer records the minimum and maximum depth value of all fragments arriving at the same pixel, and this initial min-max depth buffer is the same as the one used in false intersection detection in Section 4.3.2.

After the initial min-max depth buffer is created GL4D proceeds to composite fragments arriving to the slice $z = z_{slice}$ in the volume frame-buffer using multiple rendering passes. In the ith rendering pass the min-max depth buffer for the next pass storing the next minimum and maximum depth values is computed, *i.e.*

$$\text{MinDepth}_i(x, y) = \min\left(\left\{w : \forall\, \text{frag}_i = (x, y, z_{slice}, w) \wedge w > \text{MinDepth}_{i-1}(x, y)\right\}\right)$$
$$\text{MaxDepth}_i(x, y) = \max\left(\left\{w : \forall\, \text{frag}_i = (x, y, z_{slice}, w) \wedge w < \text{MaxDepth}_{i-1}(x, y)\right\}\right).$$

At the same time fragments bearing the same depth value as the minimum or the maximum depth value in the min-max depth buffer $(\text{MinDepth}_{i-1}, \text{MaxDepth}_{i-1})$ is rendered and composited with fragments from previous passes. The `GL_MAX` blending equation is used for min-max depth buffer generation and front-to-back composition and the conventional `GL_SRC_ALPHA` and `GL_ONE_MINUS_SRC_ALPHA` is used for back-to-front composition.

The front-to-back composition equation is the same as the one used in [56]. The color C_i and alpha A_i values are associated with the fragment (x, y, z_{slice}, w_i), *i.e.* $C_0, \ldots, C_{n-1}$ and $A_0, \ldots, A_{n-1}$ are ordered in front-to-back order. C'_i and

A'_i in the equation represents the state of the composition buffer after compositing color (C_i, A_i).

$$\begin{aligned} C'_{-1} &= 0 \\ A'_{-1} &= 1 \\ C'_i &= A'_{i-1}(A_i C_i) + C'_{i-1} \\ A'_i &= (1 - A_i)A'_{i-1} \end{aligned}$$

Furthermore, since we are using `GL_MAX` blending we need to make C'_i and A'_i monotonically increasing for increasing i. This is achieved by tweaking the above equations to write $1 - A'_i$ instead of A'_i to the composition buffer.

$$\begin{aligned} C''_{-1} &= 0 \\ A''_{-1} &= 0 \\ C''_i &= (1 - A''_{i-1})(A_i C_i) + C''_{i-1} \\ A''_i &= 1 - (1 - A_i)(1 - A''_{i-1}) \end{aligned}$$

The pseudo-code in Algorithm 3 summarizes the 4D dual depth peeling algorithm for rendering one slice in the volume frame-buffer.

4.3.4 Optimization

Various optimization techniques have been employed in GL4D to achieve interactive frame-rate without sacrificing rendering quality. This section describes a novel GPU-assisted hexahedral culling that dramatic reduces the number of tetrahedra being processed for each rendering pass by removing tetrahedra that have no intersection with the slicing plane.

GPU-assisted Hexahedral Culling

Each slice of the 3D frame-buffer is to be rendered and composited independently to produce the volume rendered image of the 3D frame-buffer. In other

Algorithm 3: The 4D dual depth peeling algorithm.

```
1  Texture MinMaxDepthTexture [2];
2  Texture FrontTexture [2];
3  Texture BackTexture;
4  Texture OutputTexture;
5  Create and initialize textures MinMaxDepthTexture, FrontTexture,
   BackTexture and OutputTexture ;
6  Initialize MinMaxDepthTexture [0] to store the minimum and maximum
   depth value for each pixel in the slice.;
7  for i ← 0 to MAX_DUAL_DEPTH_PEELING_ITERATIONS−1 do
8      Clear BackTexture ;
9      glBlendEquation(GL_MAX);
10     Shader Program 1
11         Write the next set of min-max depth value for next iteration to
           MinMaxDepthTexture [(i + 1) % 2] ;
12         Accumulate fragments from front-to-back by compositing
           fragments having the same depth value as the minimum depth
           value in MinMaxDepthTexture [i % 2] with fragment in
           FrontTexture [i % 2] using the front-to-back blending equations ;
13         Write the composited result for front-to-back composition to
           FrontTexture [(i + 1) % 2] ;
14         Write fragments having the same depth value as the maximum
           depth value in MinMaxDepthTexture [i % 2] to BackTexture ;
15     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
16     Shader Program 2
17         Composite the BackTexture to OutputTexture.;
18         Empty fragments in BackTexture is discarded to allow occlusion
           query to terminate the dual depth peeling loop when there is
           nothing more to peel ;
19 Shader Program 3
20     Composite FrontTexture to OutputTexture ;
```

words the 3-manifold have to be rendered repeatedly for each slice in a naïve implementation of the 4D rendering pipeline. Based on the observation that not all tetrahedra in the 3-manifold participate in the rendering process of one slice—only tetrahedra which intersect with the slicing plane do. We have employed an algorithm to cull these out-of-slice tetrahedra by dividing the 3-manifold into multiple patches and calculate a bounding hypercube for each patch. With the assistance of GPU these bounding hypercubes will be used to cull patches of tetrahedra mesh that do not intersect with the slicing plane.

The 3-manifold model is first preprocessed by separating it into multiple surface patches along the parametric space at regular interval. For each patch a bounding hypercube is calculated to bound the whole surface patch in $\mathbb{R}^4$. This process is akin to breaking up a 2-manifold in $\mathbb{R}^3$ into patches and calculate a bounding cube to bound each of these patches. The following equation formalizes this operation with a given surface patch P in an n-dimensional space, *i.e.* $P \in \mathbb{R}^n$, and the two extrema for the bounding hypercube

$$B = \left((min(p_x), min(p_y)), (max(p_x), max(p_y))\right) \ \forall p \in P \in \mathbb{R}^n.$$

The two extrema can be used to calculate the vertices of the bounding hypercube. A bounding hypercube in $\mathbb{E}^n$ has 2^n vertices. Algorithm 4 can be used to generate all vertices for the bounding hypercube in $\mathbb{E}^n$ and store them in GPU memory as vertex buffer objects.

Algorithm 4: Generation of all vertices for a bounding hypercube in $\mathbb{E}^n$

1 `Vertex` Hull [2];
2 Hull [0] = *first extremum*;
3 Hull [1] = *second extremum*;
4 **for** $i \leftarrow 0$ **to** $2^n - 1$ **do**
5   `Vertex` v;
6   **for** $j \leftarrow 0$ **to** $n - 1$ **do**
7         $v[j] \leftarrow \text{Hull}[(i \mathbin{\&} (1 << j)) > 0][j]$;
8   *Store v to a vertex buffer object*;

In GL4D 16 vertices are used to specify the bounding hypercube of a surface patch. Before rendering a frame the nearest and furthest depth values are computed for each bounding hypercube under the 4D transformation. The min-max depth buffer algorithm had been modified to calculate the nearest and furthest depth value for the bounding hypercubes: The 16 vertices of a bounding hypercube are transformed by vertex shader and are written to the same fragment location with the color value $(-z, z, 0, 0)$, z being the z-coordinate of a hypercube vertex after 4D transformation, with `GL_MAX` blending equation. The above procedure allows us to compute the minimum and maximum depth value for each bounding hypercube efficiently using GPU. Memory usage of the target texture is further reduced by packing the minimum and maximum depth value of two bounding hypercube into one pixel, red and green channel for one hypercube and blue and alpha channel for another, this technique effectively halves the memory usage and allows us to optimize the speed when reading back from the texture by CPU.

Chapter 5

Results

5.1 Data Sets

We have developed a series of data sets to work with the GL4D rendering platform. These data sets fall into two categories: 3- and 2-manifolds. These 3- and 2-manifolds are defined in $\mathbb{E}^4$, *i.e.* $\mathcal{M}^{3\to 4}$ and $\mathcal{M}^{2\to 4}$ respectively, but they differ in the number of degree of freedom on the surface. GL4D only supports rendering $\mathcal{M}^{3\to 4}$ as only $\mathcal{M}^{3\to 4}$ are capable of interacting with 4D light sources. A thickening operation can be used to upgrade $\mathcal{M}^{2\to 4}$ to $\mathcal{M}^{3\to 4}$ as a preprocessing step to render $\mathcal{M}^{2\to 4}$ in GL4D.

We take analogies from 3-dimensional space to illustrate this. There are two types of surfaces with different dimension in three-dimensional Euclidean space $\mathbb{E}^3$: 2-manifolds (two dimensional surfaces, $\mathcal{M}^{2\to 3}$) and 1-manifolds (one dimensional lines, $\mathcal{M}^{1\to 3}$). 2-manifolds $\mathcal{M}^{2\to 3}$ in $\mathbb{E}^3$ are capable of interacting with 3D light sources, while 1-manifolds $\mathcal{M}^{1\to 3}$ in $\mathbb{E}^3$ do not as normal vectors are not defined for a line in $\mathbb{E}^3$.

While 1-manifolds $\mathcal{M}^{1\to 3}$ do not have normal vectors we can still to define a normal plane for each point t on $\mathcal{M}^{1\to 3}(t)$. The normal plane at $\mathcal{M}^{1\to 3}(t)$ is perpendicular to the tangent vector $\frac{d\mathcal{M}^{1\to 3}}{dt}(t)$ at $\mathcal{M}^{1\to 3}(t)$. If we attach circles (1-sphere, S^1) with a fixed radius r to each point $\mathcal{M}^{1\to 3}(t)$ on the 1-manifold then the 1 dimensional line will be thickened to a tube with 2 dimensional

surface. The thickening process is essentially forming topological product $\mathcal{M}^{2\to3} = \mathcal{M}^{1\to3} \times S^1$ between $\mathcal{M}^{1\to3}$ and S^1. In general only $(n-1)$-manifolds defined in n-dimensional Euclidean space $\mathbb{E}^n$ are capable of interacting with lights, but we can thicken $(n-2)$-manifolds to $(n-1)$-manifolds by forming topological product between them and another 1-manifold such as circles (S^1).

Similarly in order to to render 2-manifolds $\mathcal{M}^{2\to4}$ in $\mathbb{E}^4$ in GL4D we need to first thicken them to $\mathcal{M}^{3\to4}$ by attaching 1-spheres to the normal plane at each point on the 2-manifolds $\mathcal{M}^{2\to4}(x, y)$. The thickened 2-manifold can be illuminated by 4D light sources and rendered by GL4D.

Most of the data sets in this section was derived from previous literatures [37], with the exception of the Fermat surface. We have reimplemented these data sets to verify the correctness of GL4D by comparing our rendered images against those generated from previous work.

5.1.1 3-manifolds in $\mathbb{E}^4$—$\mathcal{M}^{3\to4}$

3-manifolds $\mathcal{M}^{3\to4}$ are three dimensional surfaces in $\mathbb{E}^4$ that are capable of reflecting lights from 4D light sources directly. These surfaces can be directly decomposed into tetrahedral mesh and rendered by GL4D without thickening.

3-sphere—S^3

An n-sphere is a generalization of ordinary spheres to higher dimensional spaces. An n-sphere has an n dimensional surface and can be embedded into spaces with dimensions from $n+1$ onwards. A circle is a 1-sphere, an ordinary sphere is a 2-sphere, and finally a hypersphere is a 3-sphere. A hypersphere (3-sphere) is a 3-manifold that can be embedded into $\mathbb{E}^4$ and therefore can be

visualized by GL4D. The equations

$$S^3(s,t,u) = \begin{cases} \cos(s) \\ \sin(s)\sin(t)\sin(u) \\ \sin(s)\sin(t)\cos(u) \\ \sin(s)\cos(t) \end{cases} \text{, where } 0 \leq s < \pi, 0 \leq t < \pi, 0 \leq u < 2\pi$$

characterize a unit 3-sphere S^3 in $\mathbb{E}^4$.

The above equations define both the position and the normal for each point on the surface of the 3-sphere.

Hypercube

The surface of a 4D hypercube is a 3-manifold formed by 8 3D cubes. The 4D hypercube can be generated directly using Algorithm 5.

Each of the eight cubes in the 4D hypercube can be directly decomposed into tetrahedra. A 4D hypercube can be decomposed into $8 \times 5 = 40$ tetrahedra if 5-tetrahedra decomposition is used. Each of the 8 cubes can be rendered using different colors to better visualize the relationship among them during 4D rotation. Figure 5.1 is a hypercube rendered with 4D perspective projection and 4D transparency but without back-face culling to achieve the classical cube-in-cube effect for hypercube visualization. Figure 5.2 contains a 4D rotation sequence of 4D hypercube rendered by GL4D.

5.1.2 2-manifolds in $\mathbb{E}^4$—$\mathcal{M}^{2\to4}$

2-manifolds $\mathcal{M}^{2\to4}$ must first be converted to 3-manifolds $\mathcal{M}^{3\to4}$ by a thickening algorithm before they can be rendered by GL4D.

Algorithm 5: Generation of a hypercube

```
1  for i ← 0 to 4 − 1 do
       // For each dimension
2      for j ← 0 to 2 − 1 do
           // For the two cubes along each dimension
3          if j = 0 then
4              ExtraDimension ← −1;
5          else
6              ExtraDimension ← 1;
7          for Cube[0] ← 0 to 2 − 1 do
8              for Cube[1] ← 0 to 2 − 1 do
9                  for Cube[2] ← 0 to 2 − 1 do
10                     Vertex v;
11                     Vertex n;
12                     for k ← 0 to i − 1 do
13                         v[k] = Cube[k] * 2 − 1;
14                         if (i + j) % 2 then
15                             v[k] ← −v[k];
16                         n[k] ← 0;
17                     v[i] ← ExtraDimension; n[i] ← ExtraDimension;
18                     for k ← i + 1 to 4 − 1 do
19                         v[k] ← Cube[k − 1] * 2 − 1;
20                         if (i + j) % 2 then
21                             v[k] ← −v[k];
22                         n[k] ← 0;
23                     OutputVertex(v); OutputNormal(n);
```

Figure 5.1: Hypercube in convergent stereoscopic view.

The Thickening Algorithm

A 2-manifold $\mathcal{M}^{2\to4}(u,v)$ in $\mathbb{E}^4$ is specified by a system of four parametric equations with two parameters u and v. Two sets of partial derivatives are calculated analytically and these two sets of partial derivatives can then be used to calculate two tangent vectors $T^u(u,v)$ and $T^v(u,v)$ for each position $\mathcal{M}^{2\to4}(u,v)$.

$$T^u(u,v) = \begin{cases} \frac{\partial \mathcal{M}^{2\to4}(u,v)_x}{\partial u} \\ \frac{\partial \mathcal{M}^{2\to4}(u,v)_y}{\partial u} \\ \frac{\partial \mathcal{M}^{2\to4}(u,v)_z}{\partial u} \\ \frac{\partial \mathcal{M}^{2\to4}(u,v)_w}{\partial u} \end{cases}$$

$$T^v(u,v) = \begin{cases} \frac{\partial \mathcal{M}^{2\to4}(u,v)_x}{\partial v} \\ \frac{\partial \mathcal{M}^{2\to4}(u,v)_y}{\partial v} \\ \frac{\partial \mathcal{M}^{2\to4}(u,v)_z}{\partial v} \\ \frac{\partial \mathcal{M}^{2\to4}(u,v)_w}{\partial v} \end{cases}$$

The triad formed by the positional vector $\mathcal{M}^{2\to4}(u,v)$ and the two tangent

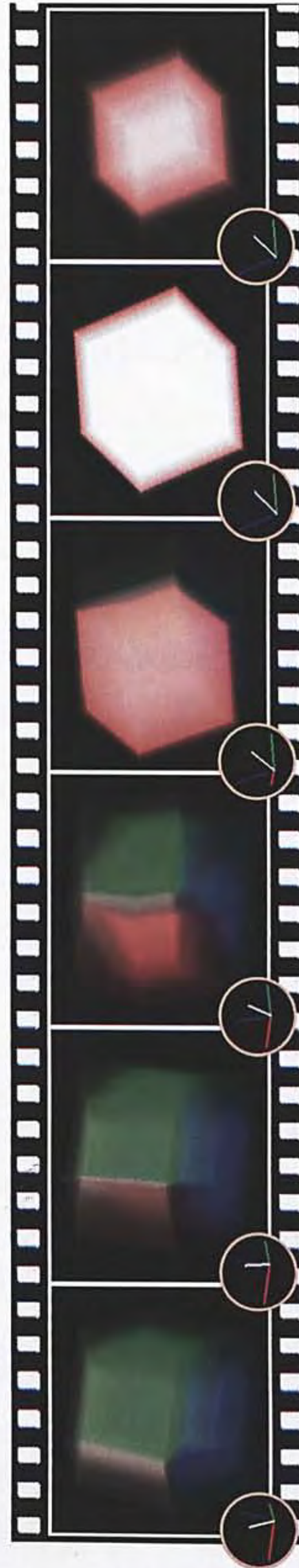

Figure 5.2: Hypercube rotation in $\mathbb{E}^4$.

vectors $T^u(u,v)$ and $T^v(u,v)$ for parameters (u,v) can be ortho-normalized by Gramm-Schmidt process to form a Cartesian frame $A(u,v)$, $B(u,v)$ and $C(u,v)$ at (u,v).

$$
\begin{aligned}
A(u,v) &= \frac{a(u,v)}{\|a(u,v)\|}; a(u,v) = \mathcal{M}^{2\to4}(u,v) \\
B(u,v) &= \frac{b(u,v)}{\|b(u,v)\|}; b(u,v) = T^u(u,v) - \text{proj}_{a(u,v)}(T^u(u,v)) \\
C(u,v) &= \frac{c(u,v)}{\|c(u,v)\|}; c(u,v) = T^v(u,v) - \text{proj}_{b(u,v)}(T^v(u,v))
\end{aligned}
$$

Two normal vectors $N^1(u,v)$ and $N^2(u,v)$ span the normal plane at position $\mathcal{M}^{2\to4}(u,v)$. The first normal vector $N^1(u,v)$ is the first vector $A(u,v)$ in the cartesian frame at $\mathcal{M}^{2\to4}(u,v)$ and the second normal vector $N^2(u,v)$ is the 4D cross product of the Cartesian frame $(A_x, A_y, A_z, A_w) = A(u,v)$, $(B_x, B_y, B_z, B_w) = B(u,v)$ and $(C_x, C_y, C_z, C_w) = C(u,v)$.

$$
\begin{aligned}
N^2(u,v) &= A(u,v) \times B(u,v) \times C(u,v) \\
&= \begin{vmatrix} i & j & k & l \\ A_x & A_y & A_z & A_w \\ B_x & B_y & B_z & B_w \\ C_x & C_y & C_z & C_w \end{vmatrix} \\
&= \begin{pmatrix} A_y(B_zC_w - B_wC_z) - A_z(B_yC_w - B_wC_y) - A_w(B_yC_z - B_zC_y) \\ A_x(B_zC_w - B_wC_z) - A_z(B_xC_w - B_wC_x) - A_w(B_xC_z - B_zC_x) \\ A_x(B_yC_w - B_wC_y) - A_y(B_xC_w - B_wC_x) - A_w(B_xC_y - B_yC_x) \\ A_x(B_yC_z - B_zC_y) - A_y(B_xC_z - B_zC_x) - A_z(B_xC_y - B_yC_x) \end{pmatrix}
\end{aligned}
$$

The 2-manifold $\mathcal{M}^{2\to4}$ can then be thickened to a 3-manifold $\mathcal{M}^{3\to4}$ by attaching a 1-spheres (circle, S^1) with radius r on the normal plane of every points on $\mathcal{M}^{2\to4}(u,v)$:

$$
\begin{aligned}
\text{normal}(\mathcal{M}^{3\to4})(u,v,\theta) &= \cos(\theta)N^1(u,v) + \sin(\theta)N^2(u,v) \\
\mathcal{M}^{3\to4}(u,v,\theta) &= \mathcal{M}^{2\to4}(u,v) + r\text{normal}(\mathcal{M}^{3\to4})(u,v,\theta).
\end{aligned}
$$

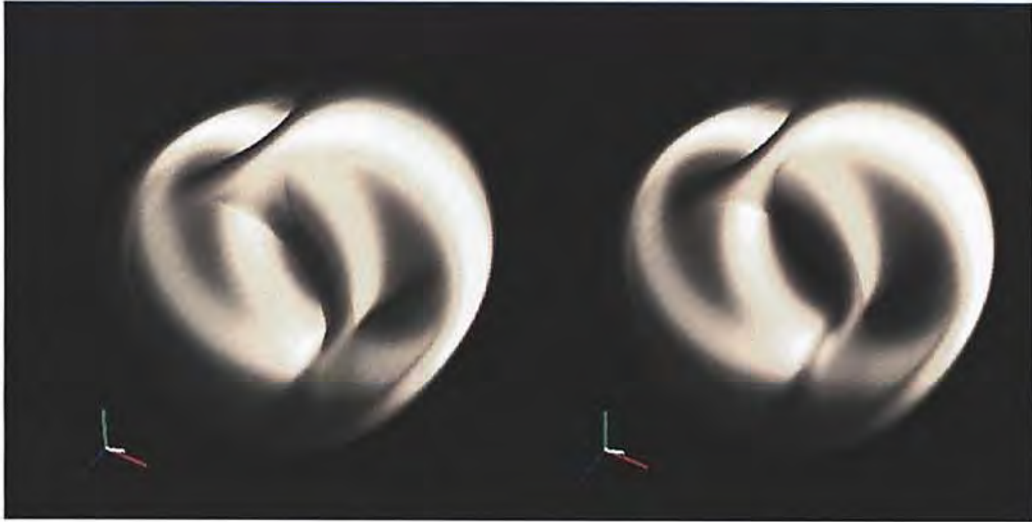

Figure 5.3: 3-torus in convergent stereoscopic view.

normal$(\mathcal{M}^{3\to4})(u,v,\theta)$ and $\mathcal{M}^{3\to4}(u,v,\theta)$ are parametric equations for normal and positional vector of the thickened 2-manifold.

3-torus—T^3

3-torus $T^3 = T^2 \times S^1 = S^1 \times S^1 \times S^1$ is a topological product of three circles (S^1). The following equations produce a 2-torus $T^2(u,v) \in \mathcal{M}^{2\to4}$ in $\mathbb{E}^4$

$$T^2(u,v) = \begin{cases} \cos(u) \\ \sin(u) \\ \cos(v) \\ \sin(v) \end{cases} \quad \text{where } 0 \le u, v < 2\pi.$$

The 2-torus $T^2(u,v)$ can be thickened to a 3-torus (Figure 5.3 and 5.4) $T^3 \in \mathcal{M}^{3\to4}(u,v)$ in $\mathbb{E}^4$ using the thickening equation.

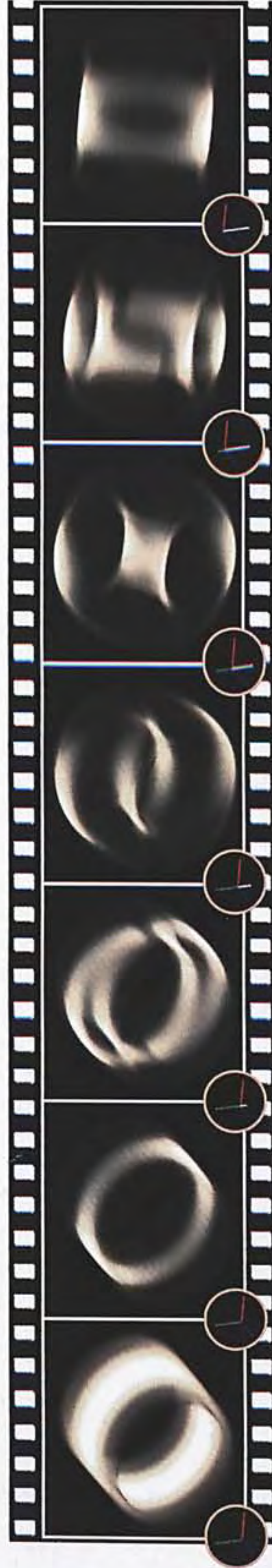

Figure 5.4: 3-torus rotation in $\mathbb{E}^4$.

Knotted Sphere

The knotted sphere starts from a Trefoil knot Trefoil(t)

$$\text{Trefoil}(t) = \begin{cases} (2+\cos 3t)\cos 2t \\ (2+\cos 3t)\sin 2t \\ \sin 3t \end{cases} \text{, where } 0 \leq t < 2\pi.$$

The Trefoil knot is then cut open and the loose ends are attached to the z-axis (Figure 5.6). The open Trefoil knot is defined piecewise with the constants

$$a = 1.414 \qquad b = \frac{a}{3} \qquad c = \frac{a}{2} \qquad d = \frac{1001}{3000} \qquad l = \frac{3}{5} \qquad x_0 = 2$$

$$P_0 = \begin{pmatrix} P_{2x} \\ 0 \\ P_{2z} \end{pmatrix} + \frac{P_{21} - P_{01}}{P_{11} - P_{01}} \left(\begin{pmatrix} P_{1x} \\ 0 \\ P_{1z} \end{pmatrix} + \begin{pmatrix} (a + b\cos(3\pi d))\cos(2\pi d) - x_0 \\ 0 \\ -c\sin(3\pi d) \end{pmatrix} \right)$$
$$+ \begin{pmatrix} 0 \\ (a + b\cos(3\pi d))\sin(2\pi d) \\ 0 \end{pmatrix}$$

$$P_1 = \begin{cases} -\left(a + b\cos\left(\frac{3\pi}{3}\right)\right)\cos\left(\frac{2\pi}{3}\right) + x_0 \\ \left(a + b\cos\left(\frac{3\pi}{3}\right)\right)\sin\left(\frac{2\pi}{3}\right) \\ c\sin\left(\frac{3\pi}{3}\right) \end{cases}$$

$$P_2 = \begin{cases} x_0 \\ \left(a + b\cos\left(\frac{3\pi}{6}\right)\right)\cos\left(\frac{2\pi}{6}\right) - \left(a + b\cos\left(3\frac{5\pi}{3}\right)\right)\left(\sin\left(2\frac{5\pi}{3}\right) + \cos\left(2\frac{5\pi}{3}\right)\right) \\ 0 \end{cases}$$

$$P_3 = \begin{pmatrix} P_{1x} \\ -0.5P_{1y} \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1.5P_{2y} \\ 1P_{2z} \end{pmatrix}$$

and the piece-wise equations

$$\text{OpenTrefoil}(s) = \begin{cases} \begin{pmatrix} \cos t P_{2x} \\ l P_{2x} \sin t + P_{2y} \\ P_{2z} \end{pmatrix}, t = \frac{(1-s)\pi}{2} & \text{if } 0 \leq s < 1 \\ \text{CatmullRom}(s-1, P_3, P_2, P_1, P_0) & \text{if } 1 \leq s < 2 \\ \begin{pmatrix} -(a + b\cos(3t))\cos(2t) + x_0 \\ (a + b\cos(3t))\sin(2t) \\ c\sin(3t) \end{pmatrix}, t = \frac{\pi}{3} + \frac{4\pi}{3}(s-2) & \text{if } 2 \leq s < 3 \\ \text{CatmullRom}\left(s-3, \begin{pmatrix} P_{0x} \\ -P_{0y} \\ -P_{0z} \end{pmatrix}, \begin{pmatrix} P_{1x} \\ -P_{1y} \\ -P_{1z} \end{pmatrix}, \begin{pmatrix} P_{2x} \\ -P_{2y} \\ -P_{2z} \end{pmatrix}, \begin{pmatrix} P_{3x} \\ -P_{3y} \\ -P_{3z} \end{pmatrix} \right) & \text{if } 3 \leq s < 4 \\ \begin{pmatrix} \cos t P_{2x} \\ l P_{2x} \sin t - P_{2y} \\ -P_{2z} \end{pmatrix}, t = \frac{(-(s-4))\pi}{2} & \text{if } 4 \leq s < 5 \end{cases}$$

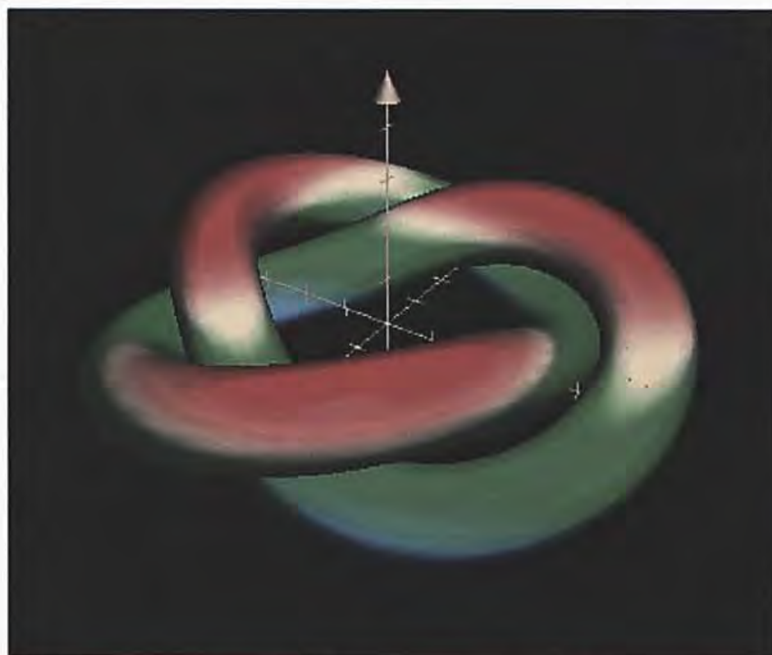

Figure 5.5: Trefoil knot in $\mathbb{E}^3$

where $0 \leq s < 5$ and Catmull-Rom spline defined by the equation

$$\text{CatmullRom}(p_0, p_1, p_2, p_3, t) = 0.5 \begin{pmatrix} t^3 \\ t^2 \\ t \\ 1 \end{pmatrix}^T \begin{pmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{pmatrix} \begin{pmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{pmatrix}.$$

The open Trefoil knot is finally spun in $\mathbb{E}^4$ to form a knotted sphere

$$\text{SpunTrefoil}(u, \theta) = \begin{cases} \text{OpenTrefoil}(u)_x \cos(\theta) \\ \text{OpenTrefoil}(u)_y \\ \text{OpenTrefoil}(u)_x \sin(\theta) \\ \text{OpenTrefoil}(u)_z \end{cases}.$$

This knotted sphere is a 2-manifold $\mathcal{M}^{2\to4}$ in $\mathbb{E}4$ and can be thickened before being rendered by GL4D (Figure 5.7 and 5.8).

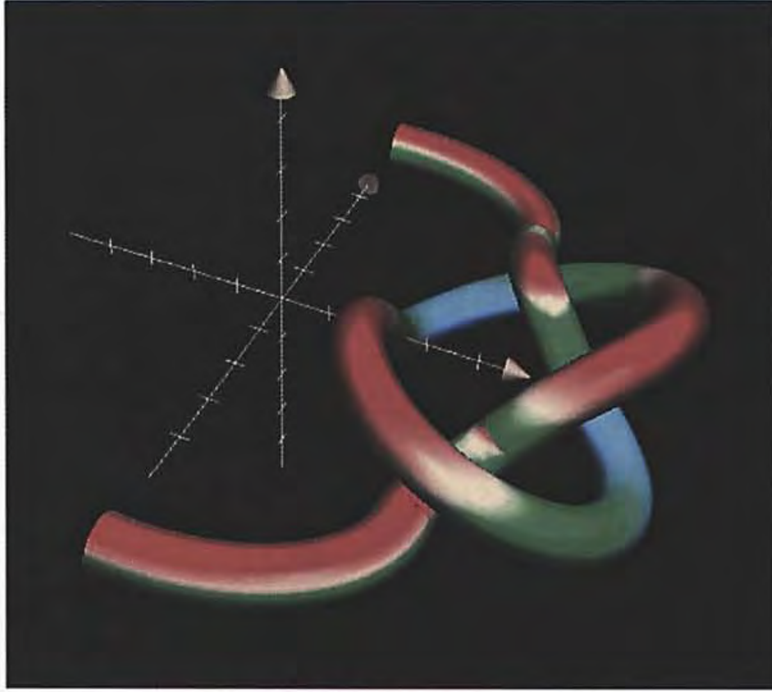

Figure 5.6: Open trefoil knot in $\mathbb{E}^3$ before spinning.

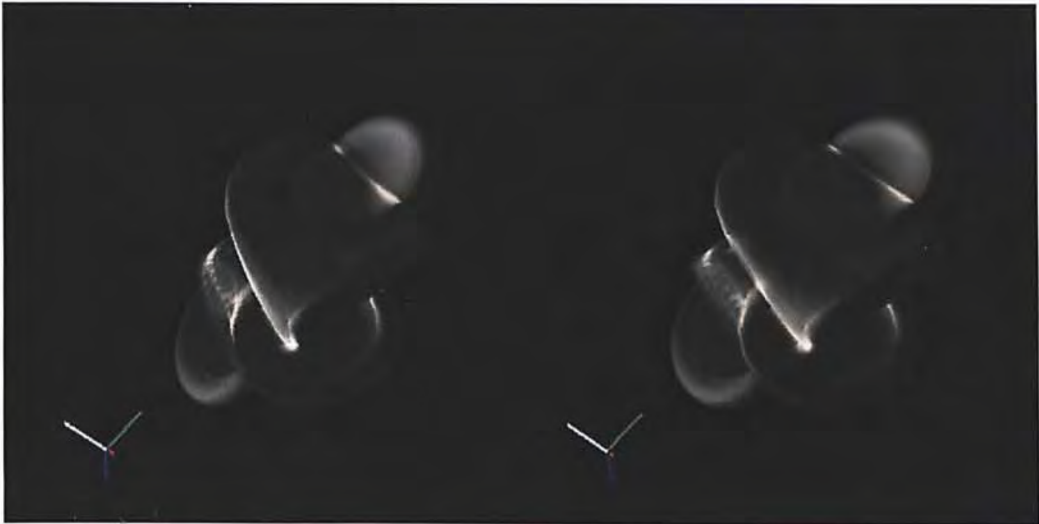

Figure 5.7: Trefoil knot hidden in knotted sphere in convergent stereoscopic view.

Figure 5.8: Knotted sphere rotation in $\mathbb{E}^4$.

Steiner Surface

A real projective plane is formed by gluing the antipodal points on the only edge that loops around a Möbius strip. Steiner surface is the immersion of real projective plane in $\mathbb{E}^4$. The equation

$$\text{Steiner}(u,v) = \begin{cases} \cos^2(u)\cos^2(v) - \sin^2(u)\cos^2(v) \\ \sin(u)\cos(u)\cos^2(v) \\ \cos(u)\sin(v)\cos(v) \\ \sin(u)\sin(v)\cos(v) \end{cases} \quad \text{where } 0 \leq u, v \leq \pi$$

describes the Steiner surface in $\mathbb{E}^4$. Since it is formed from a Möbius strip it is a one-sided surface. Figure 4.6 is a stereoscopic rendition of the Steiner surface.

Fermat Surfaces

(n_1, n_2)-Fermat surface (Fermat(n_1, n_2)) is a 2-manifold in E^4 formed by a patch work of $n_1 \times n_2$ 2-manifolds (FermatPatch$(k_1, n_1; k_2, n_2)$) [57].

$$a(\phi) = \frac{1}{2}(e^{\phi} + e^{-\phi})$$
$$b(\phi) = \frac{1}{2}(e^{\phi} - e^{-\phi})$$

$$u_{1r}(\theta, \phi) = \sqrt[n_1]{(a(\phi)\cos\theta)^2 + (b(\phi)\sin\theta)^2}$$
$$u_{1\psi}(\theta, \phi) = \frac{2}{n_1}\tan^{-1}\left(\frac{b(\phi)\sin\theta}{a(\phi)\cos\theta}\right)$$

$$u_{2r}(\theta, \phi) = \sqrt[n_2]{\left(a(\phi)\cos\left(\frac{\pi}{2} - \theta\right)\right)^2 + \left(b(\phi)\sin\left(\frac{\pi}{2} - \theta\right)\right)^2}$$
$$u_{2\psi}(\theta, \phi) = \frac{2}{n_2}\tan^{-1}\left(-\frac{b(\phi)\sin\left(\frac{\pi}{2} - \theta\right)}{a(\phi)\cos\left(\frac{\pi}{2} - \theta\right)}\right)$$

$$\text{FermatPatch}(k_1, n_1; k_2, n_2) = \begin{cases} u_{1r}(\theta, \phi)\cos\left(2\pi\frac{k_1}{n_1} + u_{1\psi}(\theta, \phi)\right) \\ u_{1r}(\theta, \phi)\sin\left(2\pi\frac{k_1}{n_1} + u_{1\psi}(\theta, \phi)\right) \\ u_{2r}(\theta, \phi)\cos\left(2\pi\frac{k_2}{n_2} + u_{2\psi}(\theta, \phi)\right) \\ u_{2r}(\theta, \phi)\sin\left(2\pi\frac{k_2}{n_2} + u_{2\psi}(\theta, \phi)\right) \end{cases},$$
$$\text{where } 0 \le \theta < \frac{1}{2}\pi, -1 \le \phi < 1$$

$$\text{Fermat}(n_1, n_2) = \bigcup_{k_1 \in \{0,1,\ldots,n_1-1\}} \bigcup_{k_2 \in \{0,1,\ldots,n_2-1\}} \text{FermatPatch}(k_1, n_1; k_2, n_2)$$

A (n_1, n_2)-Fermat surface can be thickened to a 3-manifold $\mathcal{M}^{3\to 4}$ and rendered in GL4D. Figure 5.9–5.17 shows a gallery of (n_1, n_2)-Fermat surfaces in convergent stereoscopic rendering.

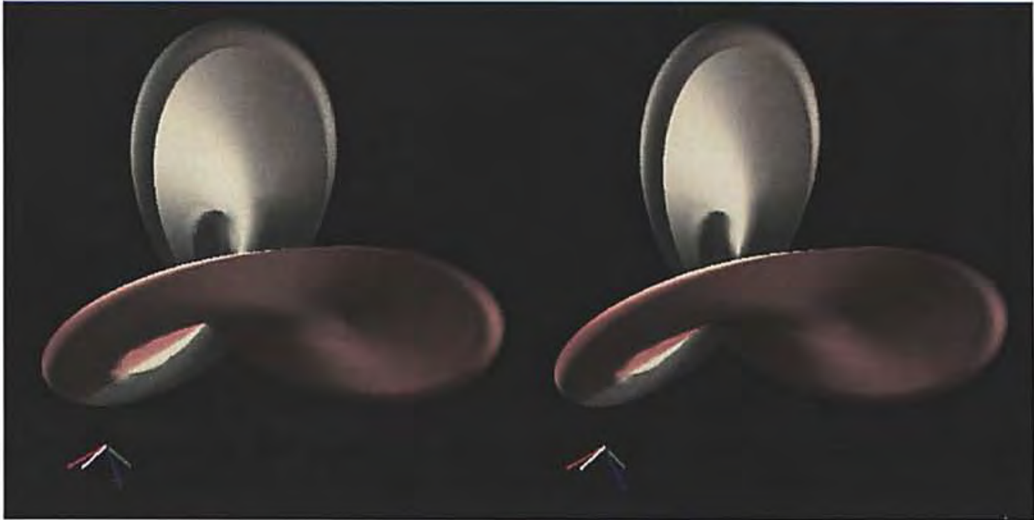

Figure 5.9: $(1,2)$-Fermat surface in convergent stereoscopic rendering.

Figure 5.10: $(1,3)$-Fermat surface in convergent stereoscopic rendering.

Figure 5.11: $(2,2)$-Fermat surface in convergent stereoscopic rendering.

Figure 5.12: $(2,3)$-Fermat surface in convergent stereoscopic rendering.

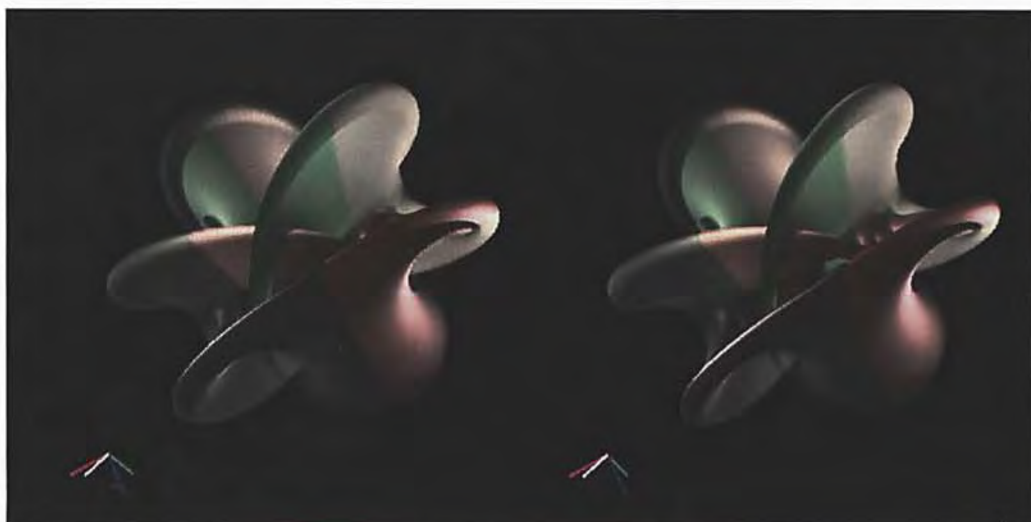

Figure 5.13: (2, 4)-Fermat surface in convergent stereoscopic rendering.

Figure 5.14: (3, 3)-Fermat surface in convergent stereoscopic rendering.

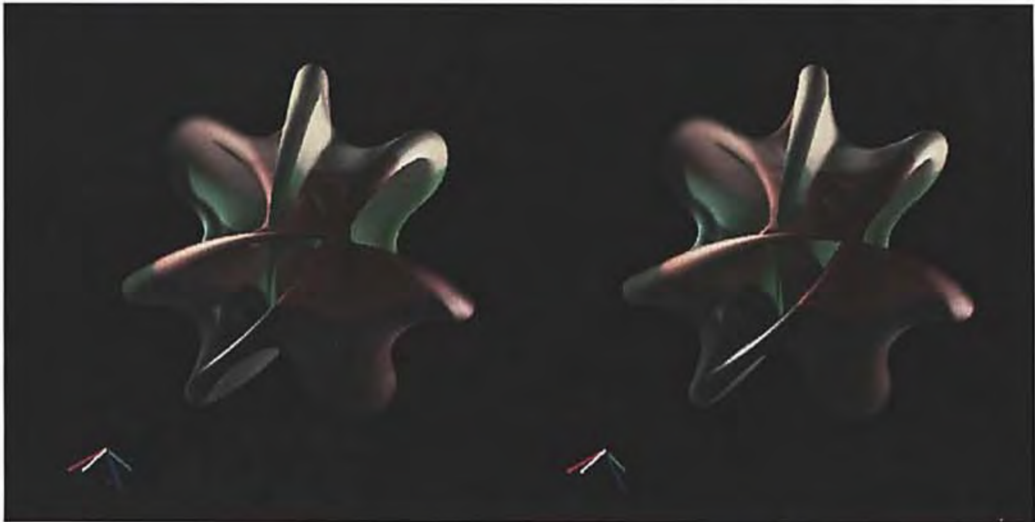

Figure 5.15: (3, 4)-Fermat surface in convergent stereoscopic rendering.

Figure 5.16: (4, 4)-Fermat surface in convergent stereoscopic rendering.

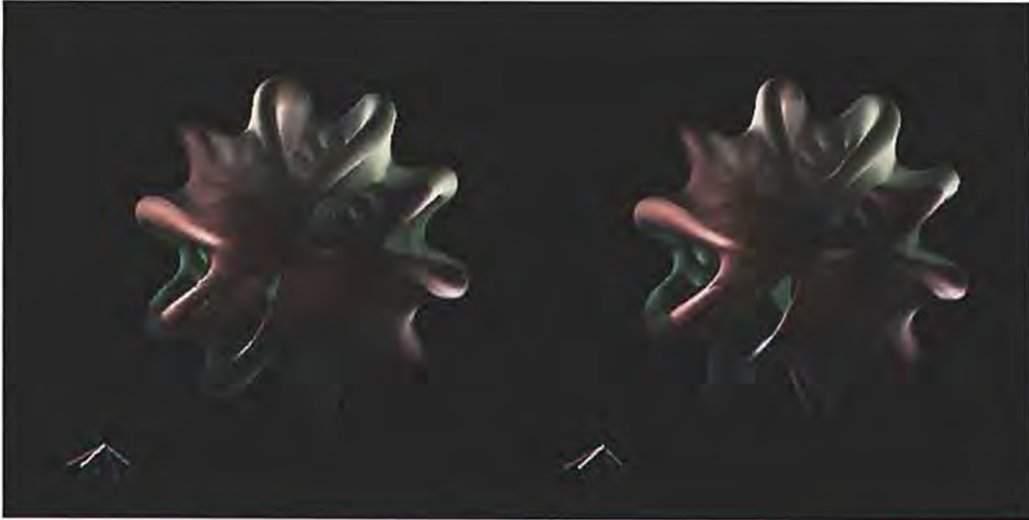

Figure 5.17: (5, 5)-Fermat surface in convergent stereoscopic rendering.

5.2 Performance

Three PC systems with different graphics hardware configurations were employed to render three opaque 4D models—hypercube, 3-torus, and Steiner surface—with 4D depth testing and per-voxel shading in 4D. Table 5.1 summarizes a performance analysis of GL4D.

- *8600 GTS*: Dell OptiPlex GX620 with Intel Pentium D 3GHz, 1GB memory, and NVIDIA GeForce 8600 GTS;
- *9800 GT*: Dell XPS 730 with Intel Core 2 Quad Q9400 2.66GHz, 3GB memory, and NVIDIA GeForce 9800 GT;
- *GTX 285*: Dell Precision T5400 with Intel Xeon 2.50 GHz, 8GB memory, and NVIDIA GeForce GTX 285.

It is worth noting that although the hypercube only has 40 4D tetrahedra, these tetrahedra are relatively large in size when compared to tetrahedra in other models; hence, they produce a substantial number of tetrahedron-slice intersections and voxel fragments. The number of slices in the 3D rasterization

	Number of slices	64	128	256	512
Hypercube	Tetrahedra sliced	1920	3840	7680	15360
(40 tetrahedra)	8600 GTS	57	30	15	8.5
	9800 GT	59.9	59.88	29.95	15.98
	GTX 285	59.95	59.95	29.95	19.98
4D Torus	Tetrahedra sliced	672000	1363200	2736000	5500800
(115200 tetrahedra)	8600 GTS	20	12.5	7.2	3.7
	9800 GT	29.95	19.98	9.98	5.44
	GTX 285	59.95	29.97	29.93	14.98
Steiner Surface	Tetrahedra sliced	886800	1770400	3533600	7064000
(115200 tetrahedra)	8600 GTS	18.6	10	5.3	2.7
	9800 GT	29.94	19.96	9.98	4.99
	GTX 285	58.53	29.97	19.97	14.98

Table 5.1: Frame-rate (frame per second) of GL4D for different 4D models and different hardware configurations with different numbers of slices.

(or voxelization) process can greatly affect the performance (and quality) of GL4D; the greater the number of slices, the more tetrahedron-slice intersections occur (the first data row for each model shown in the table), and hence, the more calls to the geometry shader and the more voxel fragments for the fragment shader to process. In general, 256 slices are employed in practice. We tested the performance of GL4D on a series of three successive generations of graphics cards: NVIDIA GeForce 8600, NIVIDIA GeForce 9800 GT, and NVIDIA GeForce GTX 285. We can see from the table that real-time performance can be achieved with the latest GPU technology. For instance, using the GeForce GTX 285 to display the flat torus using 256 slices, GL4D can generate 81.9M tetrahedron-slice intersections per second.

Chapter 6

Conclusion

GL4D is a 4D rendering platform which includes utility programs to generate and preprocess manifolds in 4D Euclidean space—which can be native 3-manifolds $\mathcal{M}^{3\to 4}$ or 2-manifold $\mathcal{M}^{2\to 4}$ after thickening—into hexahedral mesh and subsequently to tetrahedral mesh. The resulting tetrahedral mesh is uploaded to memory on graphics processor as OpenGL vertex buffer objects (VBO). The core GL4D rendering pipeline can then rasterizes the uploaded tetrahedral mesh to a 3D frame-buffer and finally volume render the frame-buffer to the 2D computer display. Advanced rendering and visualization techniques such as stereoscopic rendering, false intersection detection and transparent 4D object rendering had also been implemented and explored in GL4D. Finally GL4D achieves interactive frame-rate and produces high-quality rendering by employing various optimization techniques and implementation tricks. These features had allowed GL4D platform to unravels the mysterious 4D objects within 4D world before us and fuel future scientific and mathematical researches.

Chapter 7

Future Work

While GL4D has a complete suite of tools for generating and rendering 4D objects from mathematical equations to computer display, much can be done to make the visualization platform perfect. Currently GL4D is capable of rendering 4D objects that had been pre-generated offline and cached in GPU memory to achieve interactive performance, and this limitation makes GL4D unsuitable for applications that requires online model editing such as CAD for 4D objects. Furthermore, the process of creating a 4D tetrahedral mesh from mathematical equations is very labor intensive and involves a lot of human intervention, *e.g.* deriving the tangential equations analytically and fixing any singularity points encountered. This non-automated model generation process made creating new models for GL4D difficult.

The GL4D visualization system is just the first step towards a more full fledged interactive 4D visualization for exploring high dimensional objects, and a lot of work is still necessary to bring 4D visualization to its full potential. Possible future research directions include designing additional visual cue to aid interpretation of 4D objects; explore the application of non-photorealistic rendering techniques to selectively emphasize important landmarks on 3-manifolds; devise a more systematic way to help users to develop intuition for high dimensional objects; and comparative study to compare between various visualization cues.

These further researches will evolve GL4D from merely a 4D rendering system to a comprehensive suite for high dimensional object exploration. These future research work will make the fruit of this research seeds new researches in the future.

Bibliography

[1] Khronos Group, "OpenGL - the industry standard for high performance graphics." http://opengl.org/.

[2] D. Luebke, M. Harris, J. Krüger, T. Purcell, N. Govindaraju, I. Buck, C. Woolley, and A. Lefohn, "Gpgpu: general purpose computation on graphics hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, (New York, NY, USA), p. 33, ACM, 2004.

[3] K. A. Mark Segal, "The OpenGL graphics system: A specification (version 4.0 (core profile) - march 11, 2010)." http://www.opengl.org/registry/doc/glspec40.core.20100311.pdf.

[4] M. Meißner, U. Hoffmann, and W. Straßer, "Enabling classification and shading for 3d texture mapping based volume rendering using opengl and extensions," in *VIS '99: Proceedings of the conference on Visualization '99*, (Los Alamitos, CA, USA), pp. 207–214, IEEE Computer Society Press, 1999.

[5] W. E. Lorensen and H. E. Cline, "Marching cubes: A high resolution 3d surface construction algorithm," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 163–169, 1987.

[6] M. Levoy, "Display of surfaces from volume data," *IEEE Comput. Graph. Appl.*, vol. 8, no. 3, pp. 29–37, 1988.

[7] H. Pfister, B. Lorensen, C. Bajaj, G. Kindlmann, W. Schroeder, L. S. Avila, K. Martin, R. Machiraju, and J. Lee, "The transfer function bake-off," *IEEE Comput. Graph. Appl.*, vol. 21, no. 3, pp. 16–22, 2001.

[8] D. Ebert and P. Rheingans, "Volume illustration: non-photorealistic rendering of volume models," in *VIS '00: Proceedings of the conference on Visualization '00*, (Los Alamitos, CA, USA), pp. 195–202, IEEE Computer Society Press, 2000.

[9] P. Rheingans and D. Ebert, "Volume illustration: Nonphotorealistic rendering of volume models," *IEEE Transactions on Visualization and Computer Graphics*, vol. 7, no. 3, pp. 253–264, 2001.

[10] S. Grimm, S. Bruckner, A. Kanitsar, and E. Gröller, "Memory efficient acceleration structures and techniques for cpu-based volume raycasting of large data," in *VV '04: Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics*, (Washington, DC, USA), pp. 1–8, IEEE Computer Society, 2004.

[11] D. S. Ebert, C. J. Morris, P. Rheingans, and T. S. Yoo, "Designing effective transfer functions for volume rendering from photographic volumes," *IEEE Transactions on Visualization and Computer Graphics*, vol. 8, no. 2, pp. 183–197, 2002.

[12] J. D. Foley, A. van Dam, and J. F. Feiner, Steven K. a nd Hughes, *Computer Graphics: Principles and Practice in C.* Addison-Wesley Professional, second ed., August 1995.

[13] C. Johnson and C. Hansen, *Visualization Handbook.* Orlando, FL, USA: Academic Press, Inc., 2004.

[14] P. Lacroute and M. Levoy, "Fast volume rendering using a shear-warp factorization of the viewing transformation," in *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, (New York, NY, USA), pp. 451–458, ACM, 1994.

[15] C. T. Silva and J. S. B. Mitchell, "The lazy sweep ray casting algorithm for rendering irregular grids," *IEEE Transactions on Visualization and Computer Graphics*, vol. 3, pp. 142–157, 1997.

[16] P. Shirley and A. Tuchman, "A polygonal approximation to direct scalar volume rendering," *SIGGRAPH Comput. Graph.*, vol. 24, no. 5, pp. 63–70, 1990.

[17] S. Röttger, M. Kraus, and T. Ertl, "Hardware-accelerated volume and isosurface rendering based on cell-projection," in *VIS '00: Proceedings of the conference on Visualization '00*, (Los Alamitos, CA, USA), pp. 109–116, IEEE Computer Society Press, 2000.

[18] R. Westermann and T. Ertl, "The vsbuffer: visibility ordering of unstructured volume primitives by polygon drawing," in *VIS '97: Proceedings of the 8th conference on Visualization '97*, (Los Alamitos, CA, USA), pp. 35–ff., IEEE Computer Society Press, 1997.

[19] D. M. Reed, R. Yagel, A. Law, P.-W. Shin, and N. Shareef, "Hardware assisted volume rendering of unstructured grids by incremental slicing," in *VVS '96: Proceedings of the 1996 symposium on Volume visualization*, (Piscataway, NJ, USA), pp. 55–ff., IEEE Press, 1996.

[20] M. Levoy, "Efficient ray tracing of volume data," *ACM Trans. Graph.*, vol. 9, no. 3, pp. 245–261, 1990.

[21] T. J. Cullip and U. Neumann, "Accelerating volume reconstruction with 3d texture hardware," tech. rep., University of North Carolina at Chapel Hill, Chapel Hill, NC, USA, 1994.

[22] M. Meißner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis, "A practical evaluation of popular volume rendering algorithms," in *VVS '00: Proceedings of the 2000 IEEE symposium on Volume visualization*, (New York, NY, USA), pp. 81–90, ACM, 2000.

[23] J. Krüger and R. Westermann, "Acceleration techniques for gpu-based volume rendering," in *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, (Washington, DC, USA), p. 38, IEEE Computer Society, 2003.

[24] B. Wylie, K. Moreland, L. A. Fisk, and P. Crossno, "Tetrahedral projection using vertex shaders," in *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, (Piscataway, NJ, USA), pp. 7–12, IEEE Press, 2002.

[25] A. M. Noll, "A computer technique for displaying N-dimensional hyperobjects," *Communication ACM*, vol. 10, no. 8, pp. 469–473, 1967.

[26] T. F. Banchoff, "Visualizing two-dimensional phenomena in four-dimensional space: A computer graphics approach," in *Statistical Image Processing and Computer Graphics* (E. Wegman and D. Priest, eds.), pp. 187–202, New York: Marcel Dekker, Inc., 1986.

[27] T. F. Banchoff, "Beyond the third dimension: Geometry, computer graphics, and higher dimensions," *Scientific American Library*, 1990.

[28] A. R. Forsyth, *Geometry of Four Dimensions*. Cambridge University Press, 1930.

[29] D. Hilbert and S. Cohn-Vossen, *Geometry and the Imagination.* New York: Chelsea, 1952.

[30] E. A. Abbott, *Flatland.* Dover Publications, Inc., 1952.

[31] G. K. Francis, *A Topological Picturebook.* Springer Verlag, 1987.

[32] S. Hollasch, "Four-space visualization of 4D objects," 1991. Master thesis, Arizona State University.

[33] C. M. Hoffmann and J. Zhou, "Some techniques for visualizing surfaces in four-dimensional space," *Computer Aided Design*, vol. 23, no. 1, pp. 83–91, 1991.

[34] S. A. Carey, R. P. Burton, and D. M. Campbell, "Shades of a higher dimension," *Computer Graphics World*, pp. 93–94, October 1987.

[35] K. V. Steiner and R. P. Burton, "Hidden volumes: The 4th dimension," *Computer Graphics World*, pp. 71–74, February 1987.

[36] A. J. Hanson and P. A. Heng, "Visualizing the fourth dimension using geometry and light," in *Proc. of IEEE Visualization '91*, pp. 321–328, 1991.

[37] A. J. Hanson and P. A. Heng, "Illuminating the fourth dimension," *IEEE Computer Graphics and Applications*, vol. 12, pp. 54–62, July 1992.

[38] A. J. Hanson, "Geometry for N-dimensional graphics," in *Graphics Gems IV* (P. Heckbert, ed.), pp. 149–170, Cambridge, MA: Academic Press, 1994.

[39] D. C. Banks, "Interactive display and manipulation of two-dimensional surfaces in four dimensional space," in *Symposium on Interactive 3D Graphics*, (New York), pp. 197–207, ACM, 1992.

[40] D. C. Banks, "Illumination in diverse codimensions," in *Computer Graphics*, pp. 327–334, 1994. SIGGRAPH 1994.

[41] A. J. Hanson and R. A. Cross, "Interactive visualization methods for four dimensions," in *Proc. of IEEE Visualization 1993*, pp. 196–203, 1993.

[42] R. A. Cross and A. J. Hanson, "Virtual reality performance for virtual geometry," in *Proc. of IEEE Visualization 1994*, pp. 156–163, 1994.

[43] S. Feiner and C. Beshers, "Visualizing N-dimensional virtual worlds with N-vision," in *SIGGRAPH 1990*, pp. 37–38, 1990.

[44] Miller and Gavosto, "The immersive visualization probe for exploring n-dimensional spaces," *IEEE Comp. Graph. and App.*, vol. 24, no. 1, pp. 76–85, 2004.

[45] K. L. Duffin and W. A. Barrett, "Spiders: a new user interface for rotation and visualization of n-dimensional point sets," in *Proc. of IEEE Visualization 1994*, pp. 205–211, 1994.

[46] A. J. Hanson, "Rotations for N-dimensional graphics," in *Graphics Gems V* (A. Paeth, ed.), pp. 55–64, Cambridge, MA: Academic Press, 1995.

[47] A. J. Hanson, "The rolling ball," in *Graphics Gems III* (D. Kirk, ed.), pp. 51–60, Cambridge, MA: Academic Press, 1992.

[48] R. Egli, C. Petit, and N. F. Stewart, "Moving coordinate frames for representation and visualization in four dimensions," *Computers and Graphics*, vol. 20, no. 6, pp. 905–919, 1996.

[49] P. Bhaniramka, R. Wenger, and R. Crawfis, "Isosurfacing in higher dimensions," in *Proc. of IEEE Visualization 2000*, pp. 267–273, 2000.

[50] N. Neophytou and K. Mueller, "Space-time points: 4D splatting on efficient grids," in *Proc. of IEEE Symposium on Volume Visualization and Graphics*, pp. 97–106, 2002.

[51] A. J. Hanson and H. Zhang, "Multimodal exploration of the fourth dimension," in *Proc. of IEEE Visualization 2005*, pp. 263–270, 2005.

[52] H. Zhang and A. J. Hanson, "Shadow-driven 4D haptic visualization," in *Proc. of IEEE Visualization 2007*, pp. 1688–1695, 2007.

[53] P. Brown and B. Lichtenbelt, "Ext_geometry_shader4 extension specification," 2007. http://developer.download.nvidia.com/opengl/specs/GL_EXT_geometry_shader4.txt (last modified: May 2007).

[54] Y. Zhou, W. Chen., and Z. Tang, "An elaborate ambiguity detection method for constructing isosurfaces within tetrahedral meshes," *Computers & Graphics*, vol. 19, no. 3, pp. 355–364, 1995.

[55] P. Bourke, "Calculating stereo pairs." http://local.wasp.uwa.edu.au/~pbourke/miscellaneous/stereographics/stereorender/, 1999.

[56] L. Bavoil and K. Myers, "Order independent transparency with dual depth peeling," 2008. White paper, NVidia, http://developer.download.nvidia.com/SDK/10/opengl/src/dual_depth_peeling/doc/DualDepthPeeling.pdf.

[57] A. J. Hanson, "A construction for computer visualization of certain complex curves," *Notices of the Amer. Math. Soc.*, vol. 41, no. 9, pp. 1156–1163, 1994.

CUHK Libraries
004828032