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

Alan Chu, Chi-Wing Fu, *Member, IEEE*, Andrew J. Hanson, *Member, IEEE*, and Pheng-Ann Heng, *Member, IEEE*

Abstract—This paper describes *GL4D*, an interactive system for visualizing 2-manifolds and 3-manifolds embedded in four Euclidean dimensions and illuminated by 4D light sources. It is a tetrahedron-based rendering pipeline that projects geometry into volume images, an exact parallel to the conventional triangle-based rendering pipeline for 3D graphics. Novel features include GPU-based algorithms for real-time 4D occlusion handling and transparency compositing; we thus enable a previously impossible level of quality and interactivity for exploring lit 4D objects. The 4D tetrahedrons are stored in GPU memory as vertex buffer objects, and the vertex shader is used to perform per-vertex 4D modelview transformations and 4D-to-3D projection. The geometry shader extension is utilized to slice the projected tetrahedrons and rasterize the slices into individual 2D layers of voxel fragments. Finally, the fragment shader performs per-voxel operations such as lighting and alpha blending with previously computed layers. We account for 4D voxel occlusion along the 4D-to-3D projection ray by supporting a multi-pass back-to-front fragment composition along the projection ray; to accomplish this, we exploit a new adaptation of the dual depth peeling technique to produce correct volume image data and to simultaneously render the resulting volume data using 3D transfer functions into the final 2D image. Previous CPU implementations of the rendering of 4D-embedded 3-manifolds could not perform either the 4D depth-buffered projection or manipulation of the volume-rendered image in real-time; in particular, the dual depth peeling algorithm is a novel GPU-based solution to the real-time 4D depth-buffering problem. *GL4D* is implemented as an integrated OpenGL-style API library, so that the underlying shader operations are as transparent as possible to the user.

Index Terms—Mathematical visualization, four-dimensional visualization, graphics hardware, interactive illumination.

1 Introduction

Visualizing geometric objects embedded in four-dimensional space is an interesting and challenging problem with a long history [1, 18, 25]. Since our everyday physical world is three-dimensional, we find it difficult to envision objects that are naturally defined in four dimensions. Computer graphics modeling tools, which are not constrained by the physical world, therefore provide a natural approach to rendering and interacting with high-dimensional mathematical objects and learning how to visualize their properties. The visualization environment we describe in this paper provides interactive computer tools for inspecting and exploring geometric objects defined using four-dimensional coordinate systems using generalized lighting and rendering methods; recent advances in graphics processors permit the interactive implementation both of standard but formerly-slow features such as 4D depth culling and new features such as 4D alpha blending. A basic and fundamental fact is that 3-manifolds are required to create plausible 4D analogs of 3D lighting models, so that the rendering problem itself is supplemented by a model-extension problem for surfaces. Thus typical examples of interesting basic 4D objects include the hypercube or tesseract, which is already a 3-manifold and therefore directly renderable, and the flat torus (T^2), which is a two-manifold and therefore must be thickened in some way to turn it into a renderable 3-manifold. Among the other wide classes of interesting objects that can be studied with our 4D rendering methods are complex functions of one complex variable and complex polynomial surfaces in two complex variables [20]. Among the spectrum of 4D visualization techniques that the environment described here can support at interactive frame rates we note, for example, 4D depth buffering, 4D alpha blending, 3D volume viewing of the projected 4D geometry with tunable transfer functions, stereo imaging, and a full 4D diffuse plus specular illumination model.

With the migration from fixed-pipeline rendering to programmable pipeline rendering [28, 33], the OpenGL API had undergone a radical revision, allowing the OpenGL API to exploit the massive parallel floating point computation power in the underlying graphics hardware for rendering as well as general purpose computation. One of our objectives here is thus to propose a novel architectural design, *GL4D*, specifically to exploit programmable shaders in the GPU to efficiently perform high-quality interactive 4D rendering and visualization. The *GL4D* architecture includes the following functionalities:

- Defining 4D geometry model data and managing the data transfer to the graphics hardware;
- Performing 4D modelview and projection transformations to a virtual 3D volumetric screen through volumetric rasterization with the geometry shader [9];
- Implementation of high-quality per-voxel operators in 4D, including the specification of 4D vertex normals and light sources to support lighting computations in the fragment shader;
- Order-independent rendering of opaque tetrahedron-based geometry in 4D using per-voxel occlusion/visibility computation and back-face culling in the geometry shader;
- Alpha-blended transparent 4D geometry rendering using a multi-slice multi-pass alpha composition of the projected geometry, extending the dual depth peeling technique to 4D;
- Support for helpful visual cues in the 4D rendering, including interactively controllable lighting and viewing, coloring coding schemes, false intersection highlights, and screen-door effects.

A final design feature is that the application programming interface (API) of *GL4D* is constructed in the style of an OpenGL library API, so that it can serve as a highly-transparent programming environment for developing 4D visualization applications.

Related Work. Early research on the visualization of high-dimensional geometry included the work of Noll [32] and Banchoff [2, 3], who exploited 3D computer graphics methods to display four-dimensional objects. Methods exploited in a variety of early works [1, 18, 19, 25–27] included wireframe representations, hyperplane slicing, color coding, view transformations, projection, and animation.

Extending the methods of 3D rendering by analogy to the fourth dimension, Burton et al. [10, 35] and Hanson and Heng [23] proposed

- *Alan Chu is with the Chinese University of Hong Kong; E-mail: achu@cse.cuhk.edu.hk .*
- *Chi-Wing Fu is with the Nanyang Technological University, Singapore; E-mail: cwfu@ntu.edu.sg .*
- *Andrew J. Hanson is with Indiana University, Bloomington; E-mail: hansona@indiana.edu .*
- *Pheng-Ann Heng is with the Chinese University of Hong Kong; E-mail: pheng@cse.cuhk.edu.hk .*

Manuscript received 31 March 2009; accepted 27 July 2009; posted online 11 October 2009; mailed on 5 October 2009.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org .

1077-2626/09/$25.00 © 2009 IEEE Published by the IEEE Computer Society

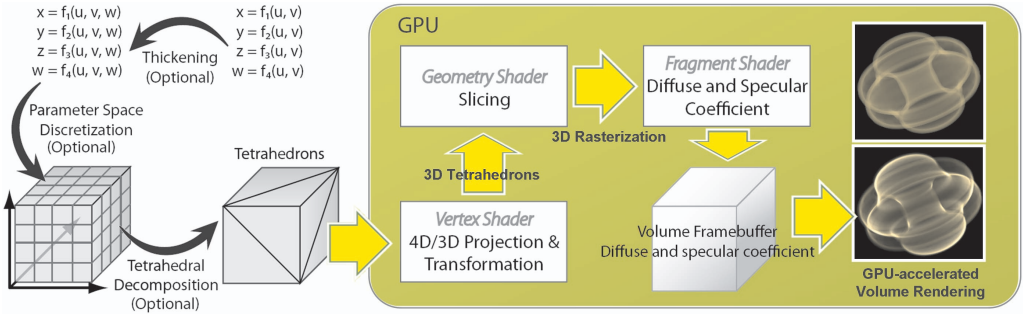

Fig. 1. Overview of the *GL4D* Architecture.

various frameworks that included lighting models for the visualization of 4D geometries. Rendering 3D geometry onto a 2D screen was replaced by projecting 4D geometry into a 3D image volume, including both color and depth buffering to support hidden surface removal in the 4D-to-3D projection. Hanson and Heng also proposed a thickening mechanism in order to convert 2D surfaces and 1D curves embedded in four-dimensional space into renderable 3-manifolds. The resulting volume images naturally required 3D volume rendering methods to expose the internal structure of the projected 4D geometry. Alpha blending along the 4D projection direction was not implemented. An alternative volume rendering to expose geometric structure after 4D-to-3D projection was suggested by Banks [4] , who employed principal curves on surfaces, transparency, and screen-door effects to highlight intersections in the projected geometry; in addition, Banks [5] proposed a general mechanism to compute diffuse and specular reflection of a k-manifold embedded in n-space. Hanson and Cross [11, 21] developed techniques implementing 4D rendering with the Shirley-Tuchman volume method [34]. assuming that the objects in 4D are static and occlusion-free in the 3D image buffer. Such methods cannot provide real-time occlusion computation and have limited interactivity compared to the methods introduced in the current paper.

Approaches to closely related problems include Feiner and Beshers [17] "worlds within worlds" interface system to manipulate and explore high-dimensional data space via nested coordinate systems. A related system developed by Miller and Gavosto [30]; used sampling methods to render and visualize 4D slices of n-dimensional data such as fractals and satellite orbits. Duffin and Barrett [12] proposed a user interface design to carry out n-dimensional rotation. Among other interesting contributions to the field are those of Egli et al. [13], who proposed a moving coordinate frame mechanism to generalize the sweeping method for representing high-dimensional data, the work of Bhaniramka et al. [8], who explored isosurfacing in high-dimensional data by a marching-cube-like algorithm for hypercubes, and that of Neophytou and Mueller [31], who investigated the use of splatting to display 4D datasets such as time-varying 3D data. Recently, Hanson and Zhang [24] proposed a multimodal user interface design that integrates visual representation and haptic interaction, allowing users to simultaneously see and touch 4D geometry; this approach was then extended [36] to exploit the idea of a reduced-dimension shadow space to directly manipulate higher-dimensional geometries.

Contributions of this paper include:

1. The *GL4D* framework, a novel visualization architecture for 4D geometry based on state-of-the-art programmable graphics hardware. Our approach carefully explores and adopts various features of the GPU, including the geometry shader, to provide the first complete example of GPU-based 4D rendering methods supporting interactive visualization.
2. The delivery of high-quality 4D geometry visualization that supports interactive controls for a wide range of aspects. This is the first environment that can deliver both high-quality voxel-based rendering of the 3D image and real-time 4D fragment lighting and blending, as well as occlusion computation for 4D geometry. In particular, we exploit a new concept of transparency processing for 4D geometry by adopting the depth peeling technique to 4D and properly blending projected fragments falling into the same voxel. In addition, we also support a variety of visual cues, such as self-intersections, within the framework of *GL4D*.
3. An OpenGL-style API library for the *GL4D* implementation that can serve as a transparent and generic interface for developing 4D visualization applications.

Paper Organization. The paper is organized as follows: After the architecture overview presented in Section 2, Sections 3 and 4 detail the GPU-based procedure for processing an input stream of 4D tetrahedrons: Section 3 focuses on the application of vertex and geometry shaders for transformations in 4D and volumetric rasterization, while Section 4 presents the per-voxel processing steps in the fragment shader, including 4D lighting and hidden surface elimination (or alpha composition in case of transparency). Section 5 shows the results of applications to various 4D geometric models and presents the library API of *GL4D*. Finally, Section 6 presents the conclusion and discusses possible directions for future work.

2 Overview: The GL4D Architecture

The *GL4D* visualization architecture is a rasterization-based rendering system for 4D virtual mathematical worlds using the tetrahedron as the rendering primitive. Its design parallels the conventional rasterization-based rendering pipeline in 3D computer graphics: In 3D computer graphics, we have 2-simplices, i.e., triangles, as the primitive elements used to represent surfaces in 3D that have unique normals and respond to lighting models; in the 4D computer graphics world of *GL4D*, we have 3-simplices, i.e., tetrahedrons, as the building blocks of hypersurfaces (3-manifolds) in 4D that have unique normals and respond to lighting models generalized to *4D virtual worlds*. Tetrahedrons are therefore the elementary rendering primitives needed to support lighting in a 4D rendering environment.

2.1 4D Geometry Input for GL4D

The *GL4D* architecture supports these tetrahedron-based inputs:

1. *Tetrahedrons* - Individual tetrahedrons are input to *GL4D* in immediate mode using the obvious sequence of `glVertex` calls between `glBegin` and `glEnd`.
2. *Hypersurfaces (Hexahedral meshes)* - A retained mode allows us to directly load hypersurfaces by means of a hexahedral grid of 4D vertices through command calls in the *GL4D API*.
3. *Surfaces embedded in 4D* - We can also input lower-dimensional geometries, such as 2-manifolds (surfaces), through command calls in *GL4D*. Just as we must thicken a 3D space curve to form a renderable tube in 3-space, we need to thicken 2-manifolds to allow them to interact uniquely with 4D lights. The thickening process in [23] is employed internally inside *GL4D* to build the geometric representations in terms of 4D tetrahedrons.

2.2 Hexahedral meshes

In 3-space, we can parameterize a surface by a mapping from a *uv*-parametric space to a 2D-grid of 3D coordinates. In 4-space, we can also parameterize an intrinsic or thickened 3-manifold by a mapping

from a *uvw*-parametric space to a 3D-grid of 4D coordinates. We refer to this geometric as a *hexahedral mesh*.

In *GL4D*, we can define a vector of four parametric equations (each for one dimension in 4D) in terms of u, v, and w, and pass the equations to *GL4D*. *GL4D* can uniformly (and possibly with an adaptive strategy) sample the equations and construct a hexahedral mesh internally. By decomposing each cell in the hexahedral mesh into tetrahedrons, *GL4D* transfers the geometry data to the tetrahedron-based processing pipeline in the GPU. It is worthwhile noting that per-vertex normals in the mesh can be computed either via an explicit parametric equation, via the Gram-Schmidt process and a 4D cross product, or explicitly input from the user.

2.3 Overview: Tetrahedron-Processing Pipeline

The tetrahedron-processing pipeline in *GL4D* is divided into the following three major components; the first one is on the CPU side, while the other two are on the GPU side; see also Figure 1.

- The first component is a tetrahedron tessellator on the CPU side to generate a tetrahedron stream given different kinds of user input, for example, hexahedral meshes or various kinds of 4D geometries.
- The second component in *GL4D* is the geometry subsystem, which parallels that in conventional 3D rendering. It includes 1) *a per-vertex transformation unit* that employs the vertex shader to transform (modelview and projection) the 4D geometry from 4D object space to 4D eye space, and finally projects to the 3D volumetric screen, and 2) *a 3D-rasterization unit* that employs the geometry shader to voxelize the input tetrahedrons into scan-planes of voxels in the 3D projected geometry.
- The last component is a GPU-based *per-fragment processing unit* that carries out per-voxel computation after 3D rasterization (or voxelization). Here we perform per-voxel operations such as lighting and occlusion computation to create a volume-rendered image of the 4D geometry. We can also include visual cues other than lighting in this per-fragment processing step; see Section 5.

3 GL4D: The Geometry Subsystem

Following the order-independent design for rendering triangles in conventional 3D graphics hardware, *GL4D* also uses order-independent processing for the input tetrahedrons. After tetrahedron transfer from the CPU side, the GPU side uses the vertex shader and the geometry shader to transform and voxelize the input tetrahedrons, and then the fragment shader processes individual voxel fragments.

3.1 Vertex Shader: Transformation and Projection

The first step on the GPU side of *GL4D* is the vertex shader for per-vertex transformation in 4D; the 4D modelview transformation is first applied to transform each vertex coordinate from object-space (input from the 4D tetrahedrons) to eye space in 4D, while the projection transformation projects the resultant eye-space coordinates to the volumetric image buffer. Users can invoke various *GL4D* commands to set up the modelview and projection transformations in 4D. Note that in 3D, we project and image 3D objects on a 2D screen, whereas in 4D, we project and image 4D objects on a volumetric screen; later in the *GL4D* pipeline, GPU-accelerated 3D volume rendering is used to represent the contents of the volumetric screen image.

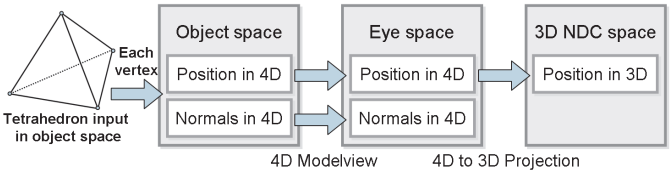

Fig. 2. Vertex shader: per-vertex processing: 4D modelview transformation followed by 4D to 3D projection.

At the end of each vertex shader call, each resultant vertex has two coordinates attached: a 4D eye coordinate after the 4D modelview transform, and a normalized device coordinate defined in the space of the volumetric screen. In addition, we also transform the 4D normal vector from the 4D object space to the 4D eye space, to support per-voxel 4D lighting later in the fragment shader; see Figure 2. Note that we can support not only orthographic projection, but also perspective projection, as demonstrated in the hypercube visualization example shown in Section 5.

3.2 Geometry shader: Rasterizing the Tetrahedrons

In order to support per-voxel lighting and per-voxel hidden surface removal (and alpha composition) in the fragment shader, we first have to rasterize the input tetrahedrons into voxel fragments inside the volumetric screen buffer. Here we apply the geometry shader provided by the programmable rendering pipeline to carry out a per-primitive (per-tetrahedron) 3D rasterization. It is worthwhile noting that existing voxelization methods [14, 16, 29] were originally designed for voxelizing triangle-based 3D models; these methods do not apply efficiently to the voxelization of our volume-bounding tetrahedrons. Our 3D rasterization of tetrahedrons is designed as follows:

Assembling Tetrahedrons. First, since no rendering primitives in conventional graphics hardware are designed for tetrahedron transfer, we must adapt existing rendering primitives so that the graphics hardware can properly assemble individual tetrahedrons in each geometry shader call. Here we employ the geometry-shader-specific primitive, namely `GL_LINES_ADJACENCY_EXT`; since this primitive type is 4-vertex-based, we can group the four vertices of each tetrahedron into a single adjacency line primitive. In this way, the primitive assembly unit in the graphics hardware can properly group the four corresponding vertices in a single geometry shader call.

Backface Culling. On commodity graphics hardware, backface culling of triangles can be done by computing the cross product of the two on-edge vectors from the first vertex in a triangle (in eye space), and then by checking the sign of the z-component in the resultant vector. If the z-component has the same sign as the viewing direction's z-component, the triangle is back-facing and can be culled.

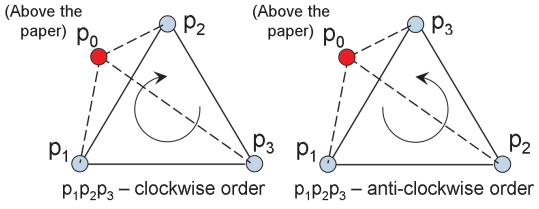

Fig. 3. Two possible vertex orderings in a tetrahedron; in the figure, p_0 is above the paper, whereas the other points are on the paper.

In 4D graphics, we can implement an analogous mechanism for discarding back-facing tetrahedrons in the geometry shader. We first must ensure that the vertices in all input tetrahedrons are ordered in a consistent manner. Note that given a 4D tetrahedron with the vertex input sequence (p_0, p_1, p_2, p_3), we can have two possible spatial arrangements as shown in Figure 3: First, we can determine a 3D subspace containing the tetrahedron similar to a 2D plane containing a triangle in 3D; then, the two possible spatial arrangements are:

1. $p_1p_2p_3$ are in clockwise order as seen from p_0 in the 3D subspace of the tetrahedron
2. $p_1p_2p_3$ are in anti-clockwise order as seen from p_0 in the 3D subspace of the tetrahedron

In *GL4D*, all input tetrahedrons should be ordered in anti-clockwise order, or else the face normal will be flipped. We can compute the 4D cross product as a determinant:

$$\text{Face normal of a tetrahedron} = \begin{vmatrix} v1_x & v2_x & v3_x & \hat{x} \\ v1_y & v2_y & v3_y & \hat{y} \\ v1_z & v2_z & v3_z & \hat{z} \\ v1_w & v2_w & v3_w & \hat{w} \end{vmatrix},$$

where $\vec{v}_i = p_i - p_0 = (vi_x, vi_y, vi_z, vi_w)$, with i = 1, 2, and 3, are the three on-edge vectors on the tetrahedron from p_0. Furthermore, since backface culling requires only the sign of the w-component in the resulting face normal, we can simplify the computation:

$$\begin{aligned} \text{Face normal}'\text{s } w-\text{component} &= \begin{vmatrix} v1_x & v2_x & v3_x \\ v1_y & v2_y & v3_y \\ v1_z & v2_z & v3_w \end{vmatrix} \\ &= v1_{xyz} \cdot (v2_{xyz} \times v3_{xyz}) . \end{aligned}$$

In this way, we can readily implement the above computation in the geometry shader as a 3D cross product followed by a 3D dot product. If the resultant w-component is negative, the tetrahedron is back-facing and can be culled. Following the convention in OpenGL, users of the *GL4D API* can also enable or disable this 4D culling feature.

Multi-slice Rasterization. In order to trigger a fragment shader call for each voxel fragment inside a 3D-rasterized tetrahedron, we employ a multi-slice (multi-pass) rasterization scheme to voxelize tetrahedrons from back to front inside the volumetric screen. In each slicing step, the tetrahedrons are voxelized on a specific slicing plane, and all these slicing planes are parallel to the 2D screen in the eye space with respect to the virtual camera that renders the volumetric screen onto the display window. Hence, we can properly voxelize each tetrahedron slice by slice from back to front in an order-independent manner, and can still correctly compose the fragment colors later in the fragment processing.

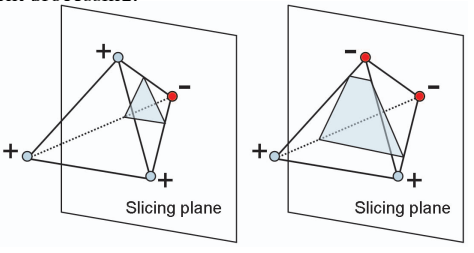

Fig. 4. Two possible cross-sections when slicing a tetrahedron with a plane: a triangle or a quadrilateral; note that vertices behind the slicing planes are colored in red.

To efficiently voxelize a tetrahedron over a specific slicing plane, we adopt the Marching Tetrahedron method [37] to rasterize a tetrahedron volume in *GL4D*. When a slicing plane intersects a tetrahedron (see Figure 4) there are only two possible voxelizable footprints: 1) a triangle, where the slicing plane cuts three edges of the tetrahedron, and 2) a quadrilateral, where the slicing plane cuts four edges of the tetrahedron. For both cases, we find that the output can be modeled as a triangle strip, and, hence, we can employ triangle strip (i.e., `GL_TRIANGLE_STRIP` in OpenGL) as the output primitive type from the geometry shader.

Moreover, our geometry shader can label each vertex as positive (takes value 1) or negative (takes value 0), depending on which side the vertex resides with respect to a given slicing plane; see also Figure 4. Then, we can pack the zeros and ones of the four vertices as a 4-bit integer so that we can quickly index the intersecting edges from a constructed edge table in the geometry shader code. In this way, we can efficiently compute the edge-plane intersections and output the intersecting shape as a triangle strip.

Output from Geometry Shader. Triangle strips output from the geometry shader are rasterized by the standard rasterization engine on the graphics hardware, and hence we can trigger a fragment shader call to process each rasterized voxel fragment.

When the geometry shader generates the triangle strips, we attach to each associated vertex a set of three attributes: a projected position (`gl_Position`) inside the volume screen buffer, and a 4D position and normal in the 4D eye space. It is worthwhile noting that a few interpolation steps are required in the geometry shader to compute these per-vertex attributes at the edge-plane intersection points because these attributes are originally given (from the vertex shader) only at the tetrahedron vertices. After the geometry shader, the standard rasterization engine can then help to further interpolate these data over the rasterized triangle strips; the fragment shader that follows thus receives these data for each voxel fragment.

4 GL4D: Voxel Fragment Processing

With the goal of supporting high-quality 4D lighting and occlusion, *GL4D* performs lighting, hidden surface removal, and transparency composition in a per-voxel manner in the fragment shader. Note that earlier 4D visualization work performed lighting and occlusion in a per-vertex or even per-primitive manner, and back-to-front sorting of tetrahedrons was required before the volume rendering. Note that when tetrahedrons are projected from 4D to 3D, the projected tetrahedrons may intersect each other in the volumetric screen region, and per-primitive sorting may not always properly identify the occluding regions. With per-voxel fragment processing, we can guarantee high-quality self-intersection detection.

4.1 Hidden surface removal

The first per-voxel-fragment operation carried out in the fragment shader is hidden surface removal. Here we take a "camouflage approach" by employing the early depth culling mechanism available on existing graphics hardware. At the end of the geometry shader, we replace the z value in the output position (e.g., the `gl_Position` output from the geometry shader) by the depth value along the 4^{th} w-dimension; hence, the early depth culling takes the w-dimension depth as its input as well as the book-keeping value in the depth buffer. As a result, we can efficiently discard occluded voxel fragments without invoking the fragment shaders on them.

4.2 Per-voxel lighting

The second per-voxel operation that takes place in the fragment shader is per-voxel lighting in 4D. Here we employ the interpolated 4D normals and positions (in eye space); these were output from the geometry shader, and were later interpolated by the standard rasterization engine. We employ the standard Phong illumination (in 4D eye space) because of its simplicity and efficiency in the shader computation:

$$I = k_a + k_d \max(\hat{\mathrm{L}} \cdot \hat{\mathrm{N}} , 0) + k_s \max((\hat{\mathrm{R}} \cdot \hat{\mathrm{V}})^{\mathrm{n_s}} , 0) ,$$

where $\hat{N}$, $\hat{V}$, $\hat{L}$, and $\hat{R}$ are the normal vector, the view vector (from the voxel fragment to view point), the light vector (from the voxel fragment to the light source), and the light-reflected vector, respectively; I is the resultant reflectance from the voxel, whereas k_a, k_d, k_s, and n_s are the ambient, diffuse, specular, and shininess terms, respectively. Note that $\hat{N}$, $\hat{V}$, $\hat{L}$, and $\hat{R}$ are all unit vectors defined in 4D eye space and 4D dot products have to be used. Furthermore, following the conventional OpenGL lighting model in local viewer mode, we can set $\hat{V}$ to be $(0,0,0,1)$ when local viewer mode is enabled.

4.3 Notion of Rendering 4D Transparent Objects

In addition to opaque 4D objects, we also support the rendering of transparent 4D objects. In the case of 3D graphics, we render transparent objects by composing the fragments that fall onto each screen pixel either from back to front or from front to back; implicitly, we base this ordering on the z-distance of the fragments from the eye point. While such an alpha composition process is well-developed in 3D graphics, its extension to 4D needs clarification.

In 4D graphics, since we eventually display the final rendering on a 2D computer screen, each screen pixel could find its projection from any point in a 2D subspace rather than along a 1D projective line as in 3D graphics. Therefore, we do not have a straightforward z ordering. We could have the following situations:

- First, if the 4D object is opaque, we can ignore the alpha composition in the w-dimension and perform the hidden surface removal as in subsection 4.1; hence, we take only the w-nearest voxel fragment in the 3D volumetric screen.

- Second, if the 4D object is transparent, we can compose the voxel fragments that fall into the same voxel (in the volumetric screen) in a back-to-front (or front-to-back) manner along the w-dimension. After alpha composition along the w-dimension, the volume rendering step that follows will further compose the voxel fragments along the z-dimension to the final screen; see Figure 5. Since the 4D modelview and projection transforms are interactively controlled by the user, we leave the subspace ordering decision to the user.

- Finally, the user can also disable the alpha composition in the volume rendering step from 3D to 2D screen, exposing only the w-dimension alpha composition effect in the nearest z layer.

Though the above approaches do not perform alpha composition directly over the entire zw-subspace for each screen pixel, we argue whether such a direct approach can exist as there are no obvious depth orderings for voxel fragments over the entire zw-subspace. Hence, we propose the above achievable approaches to rendering 4D transparent objects, and indeed, these are feasible methods that can be practically realized on shaders with existing GPU technology, as we see in the next subsection.

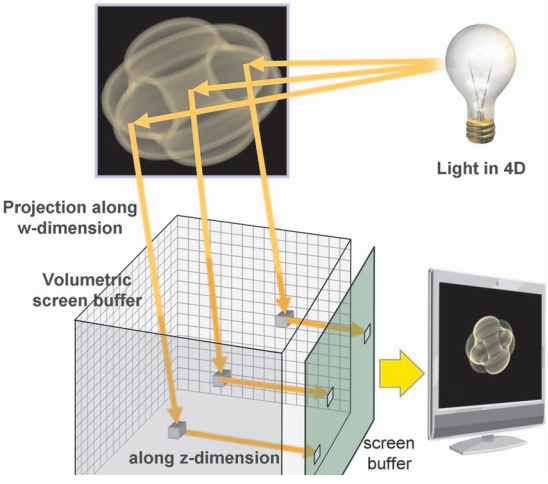

Fig. 5. Rendering transparent 4D objects: alpha composition along w-dimension followed by z-dimension.

4.4 Dual Depth Peeling for 4D Transparent Objects

To support efficient alpha composition along the w-dimension with proper depth sorting (as in the second and third approaches above), we extend the conventional dual depth peeling method [7, 15]; this method can handle $2n$ depth layers in n rendering passes, and five textures are needed in the implementation: two of them are used for storing minimum and maximum per-pixel depth values in a ping-pong manner across subsequent rendering passes; another two store per-pixel color and alpha values accumulated from the front and back peeling side (also in a ping-pong manner); the last one is for storing the color and alpha values accumulated from back to front peeling.

Note that the ping-pong technique is adopted for the min-max depth value, the accumulated color, and alpha values to avoid a read-write hazard when going from one rendering pass to the next, and we use *GL_MAX* blending mode for all three render targets as in OpenGL. With such an adaptation, we can produce transparent renderings of objects in 4D such as the 4D hypercube as illustrated in Figure 14.

5 Implementation and Results

In this section, we first describe the implementation issues and performance analysis of *GL4D*; we then present the visualization results for various 4D models and briefly outline the library API of *GL4D*.

5.1 Implementation Issues

GL4D is implemented on top of OpenGL 2.1, and requires support for the geometry shader to handle per-tetrahedron processing. The basic principles of the implementation are listed in the following.

Vertex Buffer Object. To avoid excessive geometry transfer and redundant vertex program computation (for the same vertex), the retained mode of *GL4D* caches the 4D geometry input on the GPU as (index-based) vertex buffer objects, which stores three arrays of data attributes: one for 4D object-space positions, one for 4D object-space normals (for each 4D position), and one for indices, with each set of four consecutive indices forming one tetrahedron in 4D.

Hexahedral-cell-to-tetrahedron Decomposition. In the hexahedral-cell-to-tetrahedron decomposition, we can divide a hexahedral cell into five or six tetrahedrons as depicted in Figure 6. However, if we examine the triangle patterns on matching faces of adjacent cells in the decomposition, the patterns in the six-tetrahedron decomposition can match properly without a T-join, while the five-tetrahedron decomposition apparently does not. However, we can work around this by using a five-tetrahedron decomposition that alternates two different orientations of the five-tetrahedron decomposition in a hexahedral mesh; see Figure 7. We can thus seamlessly match the diagonals across neighboring hexahedral cell faces (see Figure 7 (right)), while generating fewer tetrahedrons as compared to the six-tetrahedron decomposition.

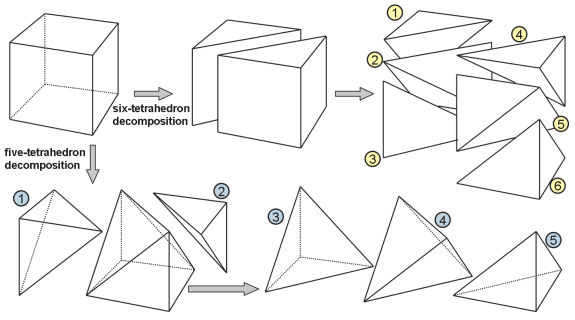

Fig. 6. Possible ways of decomposing a hexahedral cell into tetrahedrons: six tetrahedrons (top) and five tetrahedrons (bottom).

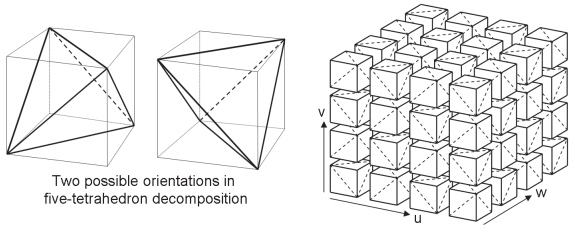

Fig. 7. Decomposing a hexahedral mesh into tetrahedrons by alternating the orientations of five-tetrahedron decompositions.

Tetrahedron-slicing. To speed up the performance of primitive assembly (also known as IA, the input assembler) and the geometry shader, we compute for each rendering frame the z range of groups of tetrahedrons. Then, for each multi-slice rendering pass (within each rendering frame), we assemble only the tetrahedrons that overlap with the z value of the current slice; hence, we can reduce the workload of primitive assembly, and avoid intensive tetrahedron-slice intersections in the geometry shader.

5.2 Performance of *GL4D*

Table 1 shows a performance analysis of *GL4D*; three PC systems equipped with different graphics boards were employed to render three different 4D models (hypercube, 4D torus, and Steiner surface) with per-voxel hidden surface removal and per-voxel lighting in 4D:

- *8600*: Dell OptiPlex GX620 with Intel Pentium D 3GHz, 1GB memory, and GeForce 8600 GTS;

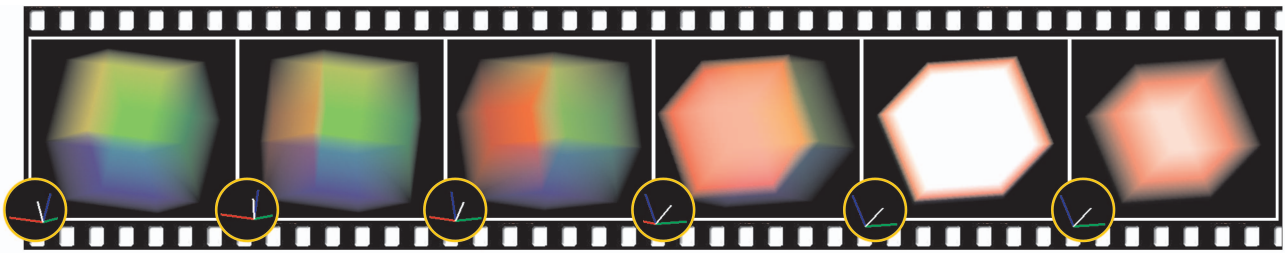

Fig. 8. Rotating the hypercube in 4D space. This is a 3-manifold composed of eight 3D cubes; we shade each 3D cube with its own color.

- *9800GT*: Dell XPS 730 with Intel Core 2 Quad CPU Q9400 2.66GHz, 3GB memory, and GeForce 9800 GT;
- *GTX285*: Dell Precision T5400 with Intel Xeon CPU 2.50 GHz, 8GB memory, and GeForce GTX285.

Table 1. Frame rate (frame per second) of *GL4D* for different 4D models and different PC systems with different numbers of slices.

	Number of slices:	64	128	256	512
Hypercube (40 tetrahedrons)	Num. Tetrahedron sliced:	1920	3840	7680	15360
	8600	57	30	15	8.5
	9800GT	59.9	59.88	29.95	15.98
	GTX285	59.95	59.95	29.95	19.98
4D Torus (115200 tetrahedrons)	Num. Tetrahedron sliced:	672000	1363200	2736000	5500800
	8600	20	12.5	7.2	3.7
	9800GT	29.95	19.98	9.98	5.44
	GTX285	59.95	29.97	29.93	14.98
Steiner surface (115200 tetrahedrons)	Num. Tetrahedron sliced:	886800	1770400	3533600	7064000
	8600	18.6	10	5.3	2.7
	9800GT	29.94	19.96	9.98	4.99
	GTX285	58.53	29.97	19.97	14.98

It is worth noting that although the hypercube only has 40 4D tetrahedrons, these tetrahedrons are relatively large in size compared to tetrahedrons in other models; hence, they produce a substantial number of tetrahedron-slice intersections and voxel fragments. The number of slices in the 3D rasterization (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 also tested the performance of *GL4D* on a series of three successive generations of graphics cards: GeForce 8600, GeForce 9800GT, and GeForce GTX285. We can see from the table that real-time performance can be achieved with the latest GPU technology. For instance, using the GTX285 to display the flat torus using 256 slices, *GL4D* can generate 81.9M tetrahedron-slice intersections per second.

5.3 Visualization Results

In this subsection, we explore and demonstrate various visualization effects on different 4D geometric models, including the hypercube, the flat torus (T^2), the knotted sphere, Steiner's Roman surface, and the CP^2 quintic 2-manifold corresponding to a cross-section of string theory's quintic Calabi-Yau 6-manifold in CP^4.

Hypercube. Considering a 3D cube as a two-manifold composed of six 2D squares bounding a solid 3D block, a 4D hypercube can similarly be thought of as a 3-manifold built from eight 3D cubes bounding a solid 4D block. Figure 8 depicts an image sequence obtained by rotating a 4D hypercube in a fixed plane in 4D space; the boundary 3D cubes are shaded with different colors so that we can see which cubes are facing the 4D camera; cubes with back-facing normals are hidden from sight in the 4D view. The initial viewpoint shows three of four possible front-facing boundary cubes and the 4D projection rotates until only the single red cube is facing the 4D camera. The lower left circles contain the four coordinate axes projected to the 2D screen space. In the penultimate view, the red axis disappears because it aligns with our 4D viewing ray. Finally, we turn down the opacity to show more internal details of the jello-like red cube.

We demonstrate 4D perspective projection in *GL4D* with the hypercube in Figure 9. The left sub-figure shows an orthographic view of the hypercube, while the middle and right figures employ shorter and shorter focal lengths (larger fields of view) in the virtual 4D pinhole camera. Unlike ordinary 3D projection, the perspective distortion in the right figure persists in *any* 3D viewpoint if we were to make a 3D rotation; it is a feature of the 4D, not the 3D, projection.

Fig. 9. Hypercube under different projections; Left: orthographic; middle: perspective; right: also perspective, but with a larger field-of-view (and the 4D camera moved closer to the hypercube).

Flat Torus. The flat torus (T^2) is a particular embedding of the familiar donut-shaped 3D torus that is completely flat in 4D Euclidean space. *GL4D* adds circles at each point to create a thickened 3-manifold from this 2-manifold embedded in 4D. Figure 10 shows an image sequence of the projected torus rotating in 4D. The initial projection to the XYZ subspace is a tube-shaped object; see the first axis icon for the first two sub-figures. After reducing the opacity value (i.e., the second sub-figure), we gradually rotate the torus to its XYW subspace. The bright rings embedded in the shape come from specular highlights in the 4D lighting. After we rotate the torus in 4D slightly off the XYZ axes (the third sub-figure), the single-ring highlights start to split and the volume visualization helps to reveal features of the internal structure as the object is rotated to different 4D projections.

Knotted Sphere. The knotted sphere embedded in 4D space is constructed by spinning a knotted line segment around a central axis. *GL4D* thickens this 2-manifold to make it locally a 3-manifold in 4-space; certain anomalies are expected to remain for topological reasons. In Figure 11, we present first the opaque knotted sphere (the leftmost sub-figure). Since the defining 2-manifold for the knotted sphere is constructed over a 2D parametric domain, *GL4D* can apply textures onto the knotted sphere in the parameter space. This opens up the interior visibility of the opaque rendering as shown in the second subfigure of Figure 11. In practice, *GL4D* can sample an input 2D texture in the fragment shader by using the parametric coordinates. Next, we can reduce the voxel opacity in the volume rendering as well as animating the lighting direction, as shown in the sequence on right hand side of Figure 11.

Steiner Surface. In the visualization of Steiner surface, we exploit the stereo viewing capability of *GL4D*. We can set up a pair of 4D virtual cameras with user-controllable interocular distance and render stereo pairs; Figure 12. shows stereo views of two different 4D orientations of the Steiner surface with divergent viewing (wall-eyed/parallel viewing). For each orientation, we render also an opaque version of the geometry.

Calabi-Yau Quintic Cross-section. Finally, we use the *GL4D* tools to render a complex patch-based surface geometry into the volumetric

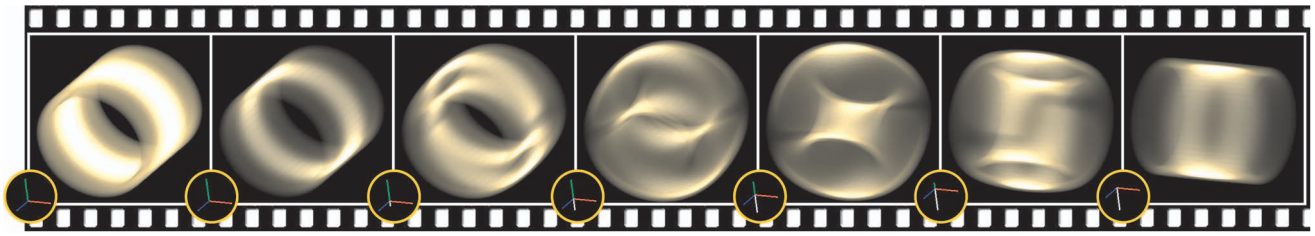

Fig. 10. Rotating the 4D embedded flat torus (from left to right) in the ZW plane, from XYZ to XYW; note the change in the axis icons.

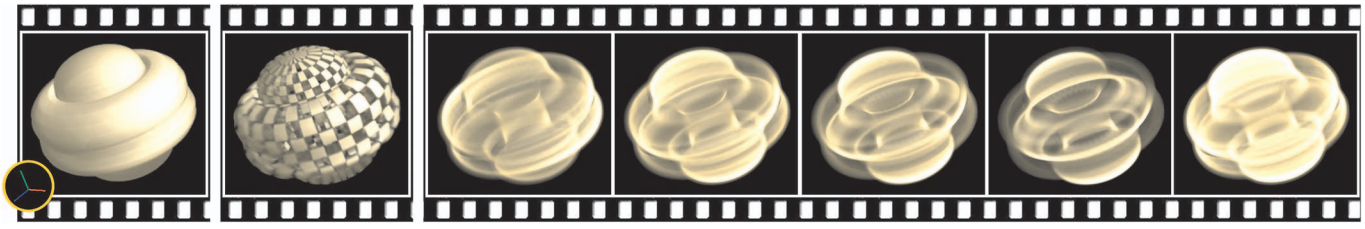

Fig. 11. Knotted sphere projected from 4D space; from left to right: we use a large opacity value in the volume rendering so that the model appears like an opaque surface; a screen door effect is added by applying a checkerboard deletion pattern in the parametric space; next, we reduce the opacity and animate the 4D lighting direction.

screen buffer. Figure 13 shows the Calabi-Yau quintic cross-section; it consists of $5^2 = 25$ patches, each shaded with different colors keyed to the two phase transformations applied to the fundamental patch [20]. Note that this complicated surface can result in a large number of self-intersections when projected to the volumetric screen buffer. *GL4D* can efficiently detect and highlight these self-intersections using a per-voxel and per-slice method, similar to the screen parallel approach to intersection curves described by Banks [6]. Intersections are marked in red in the right column of Figure 13.

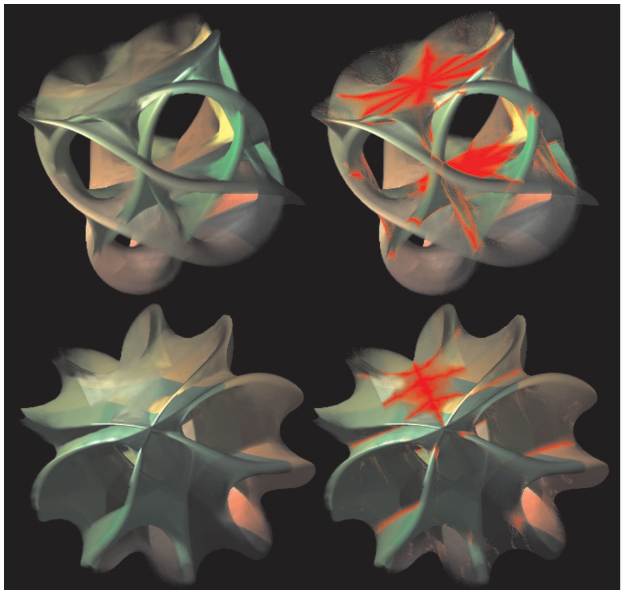

Fig. 13. Two different views of Calabi-Yau quintic cross-section in 4D; the red color (2nd column) indicates self-intersection in the projection.

4D Transparency with Dual Depth Peeling. By adapting the dual peeling method to render 4D objects, we can correctly sort and compose fragments that are projected into the same voxel location in the volume screen buffer. Figure 14 illustrates the rendering result; the first column shows two poses of the hypercube, rendered without the dual depth peeling; only one color is received per voxel as the object is opaque in the 4D to 3D projection. The second and third columns show corresponding stereo views, but with 4D transparency supported by dual depth peeling; here, each voxel can receive multiple colors originating from different boundary cubes in the projection along w.

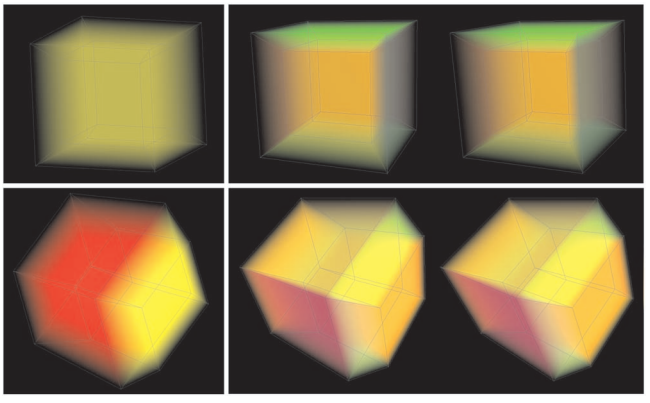

Fig. 14. Rendering the 4D hypercube with and without 4D transparency, in the right and left columns, respectively; stereo images with divergent viewing are used in the transparent renderings.

5.4 The library API

The library API of *GL4D* is built on top of OpenGL 2.1, and it consists of the following four groups of functions:

Initialization: First, we have *gl4DInit* to initialize various off-screen rendering buffers (implemented using frame buffer objects in OpenGL) and load the shader programs (vertex, geometry, and fragment shaders) into the GPU memory.

Parameter settings: Next, we can employ *gl4DParam* to assign values to various parameters in *GL4D*, including transformations, lighting, material, texture, culling, opacity values in the w-dimension and z-dimension accumulation, and various state enabling parameters.

Geometry input: There are two rendering modes in *GL4D*:

- Immediate mode: we can call *gl4DBegin* (with *GL4D_TETRAHEDRONS* or *GL4D_TETRAHEDRON_STRIP*) to start an input session, followed by some number of calls to *gl4DNormal* and *gl4DVertex* to input normals and vertices, ending with *gl4DEnd*.
- Retained mode: we can also pass the entire 4D geometry through various *GL4D* functions, e.g., *gl4DSurface* defines a parametric surface (with thickening) and *gl4DHexahedralGrid* creates a hexahedral grid of 4D vertices.

Rendering control: Finally, we use *gl4DClearBuffer* to clear the buffers , and *gl4DRenderBegin* and *gl4DRenderEnd* to define a rendering session in *GL4D*; *gl4DRenderBegin* starts a *GL4D* rendering

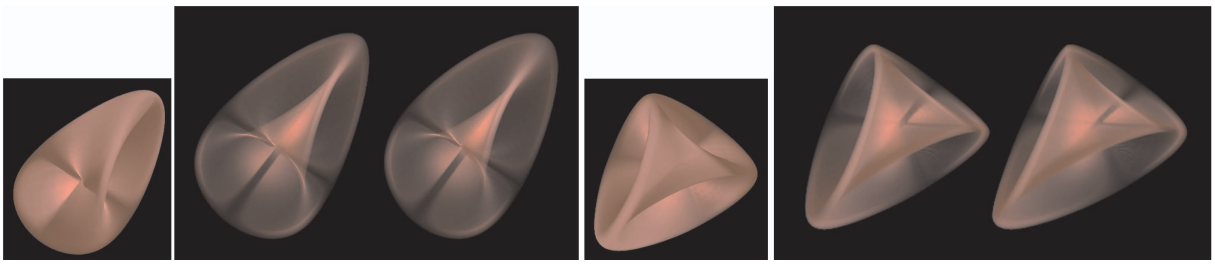

Fig. 12. Steiner surface: two stereoscopic views using divergent (wall-eyed) viewing.

session and activates the shaders, while *gl4DRenderEnd* ends the session by performing GPU-accelerated volume rendering: it renders the tetrahedron fragments that have accumulated in the volumetric buffer into the standard frame buffer.

6 Conclusion

This paper proposes a carefully-designed visualization architecture that adapts a state-of-the-art programmable rendering pipeline for the visualization of 4D mathematical models. The proposed *GL4D* architecture is a highly-efficient GPU-based API, taking advantage of all existing shader modules in the GPU hardware to efficiently process a stream of tetrahedrons and produce volume-rendered views of 4D geometries. We incorporate visual effects into the *GL4D* framework, including stereo viewing, texturing, a screendoor effect, self-intersection flags, and 4D lighting, as well as the novel notion of 4D transparency composition; the latter is supported by extending the dual depth peeling method into the fourth w-dimension.

In future work, we hope to explore the inclusion of high-performance ray tracing methods in *GL4D* to improve the rendering quality and to extend the scope to include such effects as 4D shadows. We would also like to apply the *GL4D* architecture to the visualization of more types of mathematical models and additional classes of 4D information such as 3D scalar fields [22] and time-dependent data. A 3D scalar field, for example, can be rendered as a 3-manifold of 4D height values (like a top-down view of a 2D elevation map), whose normal vectors result in extremely detailed 4D diffuse and specular shading effects in the volume rendering. We also hope to integrate *GL4D* with interaction methods such as haptic devices and the Wii Remote.

Acknowledgments

We would like to thank the anonymous reviewers, who provided invaluable comment to help the authors improve the manuscript. This research was supported in part by MOE AcRF Tier1 Grant (RG 13/08) and NSF-0430730.

References

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

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

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

[4] D. C. Banks. Interactive display and manipulation of two-dimensional surfaces in four dimensional space. In *Symposium on Interactive 3D Graphics*, pages 197–207, New York, 1992. ACM.

[5] D. C. Banks. Illumination in diverse codimensions. In *Computer Graphics*, pages 327–334, 1994. SIGGRAPH 1994.

[6] D. C. Banks. Screen-parallel determination of intersection curves. *Parallel Computing*, 23(7):953–960, 1997.

[7] 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.

[8] P. Bhaniramka, R. Wenger, and R. Crawfis. Isosurfacing in higher dimensions. In *Proc. of IEEE Visualization 2000*, pages 267–273, 2000.

[9] 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).

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

[11] R. A. Cross and A. J. Hanson. Virtual reality performance for virtual geometry. In *Proc. of IEEE Visualization 1994*, pages 156–163, 1994.

[12] 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*, pages 205–211, 1994.

[13] R. Egli, C. Petit, and N. F. Stewart. Moving coordinate frames for representation and visualization in four dimensions. *Computers and Graphics*, 20(6):905–919, 1996.

[14] E. Eisemann and X. Décoret. Fast scene voxelization and applications. In *Proc. of the 2006 symposium on Interactive 3D graphics and games*, pages 71–78, 2006.

[15] C. Everitt. Interactive order-independent transparency, 2001. White paper, NVidia, http://developer.nvidia.com/object/Interactive_Order_Transparency.html.

[16] S. Fang and H. Chen. Hardware accelerated voxelization. *Computers & Graphics*, 24(3):433–442, 2000.

[17] S. Feiner and C. Beshers. Visualizing N-dimensional virtual worlds with N-vision. In *SIGGRAPH 1990*, pages 37–38, 1990.

[18] A. R. Forsyth. *Geometry of Four Dimensions*. Cambridge U. Press, 1930.

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

[20] A. J. Hanson. A construction for computer visualization of certain complex curves. *Notices of the Amer. Math. Soc.*, 41(9):1156–1163, 1994.

[21] A. J. Hanson and R. A. Cross. Interactive visualization methods for four dimensions. In *Proc. of IEEE Visualization 1993*, pages 196–203, 1993.

[22] A. J. Hanson and P. A. Heng. Four-dimensional views of 3D scalar fields. In *Proc. of IEEE Visualization '92*, pages 84–91, 1992.

[23] A. J. Hanson and P. A. Heng. Illuminating the fourth dimension. *IEEE Computer Graphics and Applications*, 12(4):54–62, July 1992.

[24] A. J. Hanson and H. Zhang. Multimodal exploration of the fourth dimension. In *Proc. of IEEE Visualization 2005*, pages 263–270, 2005.

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

[26] C. M. Hoffmann and J. Zhou. Some techniques for visualizing surfaces in four-dimensional space. *Computer Aided Design*, 23(1):83–91, 1991.

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

[28] E. Lindholm, M. J. Kligard, and H. Moreton. A user-programmable vertex engine. In *SIGGRAPH 2001*, pages 149–158, 2001.

[29] I. Llamas. Real-time voxelization of triangle meshes on the GPU. In *ACM SIGGRAPH 2007 sketches*, page 18, 2007.

[30] Miller and Gavosto. The immersive visualization probe for exploring n-dimensional spaces. *IEEE Comp. Graph. and App.*, 24(1):76–85, 2004.

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

[32] A. M. Noll. A computer technique for displaying N-dimensional hyperobjects. *Communication ACM*, 10(8):469–473, 1967.

[33] K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *SIGGRAPH 2001*, pages 159–170, 2001.

[34] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. volume 24, pages 63–70, 1990. SIGGRAPH 1990.

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

[36] H. Zhang and A. J. Hanson. Shadow-driven 4D haptic visualization. In *Proc. of IEEE Visualization 2007*, pages 1688–1695, 2007.

[37] Y. Zhou, W. Chen., and Z. Tang. An elaborate ambiguity detection method for constructing isosurfaces within tetrahedral meshes. *Computers & Graphics*, 19(3):355–364, 1995.