Utah State University

DigitalCommons@USU

Space Dynamics Laboratory Publications Space Dynamics Laboratory

1-1-2014

# GPU-Based Volume Visualization from High-Order Finite Element Fields

Blake Nelson

Robert M. Kirby

Robert Haimes

Follow this and additional works at: https://digitalcommons.usu.edu/sdl_pubs

## Recommended Citation

Nelson, Blake; Kirby, Robert M.; and Haimes, Robert, "GPU-Based Volume Visualization from High-Order Finite Element Fields" (2014). *Space Dynamics Laboratory Publications.* Paper 101.
https://digitalcommons.usu.edu/sdl_pubs/101

This Article is brought to you for free and open access by the Space Dynamics Laboratory at DigitalCommons@USU. It has been accepted for inclusion in Space Dynamics Laboratory Publications by an authorized administrator of DigitalCommons@USU. For more information, please contact digitalcommons@usu.edu.

# GPU-Based Interactive Cut-Surface Extraction From High-Order Finite Element Fields

Blake Nelson, Robert Haimes, and Robert M. Kirby, *Member, IEEE*

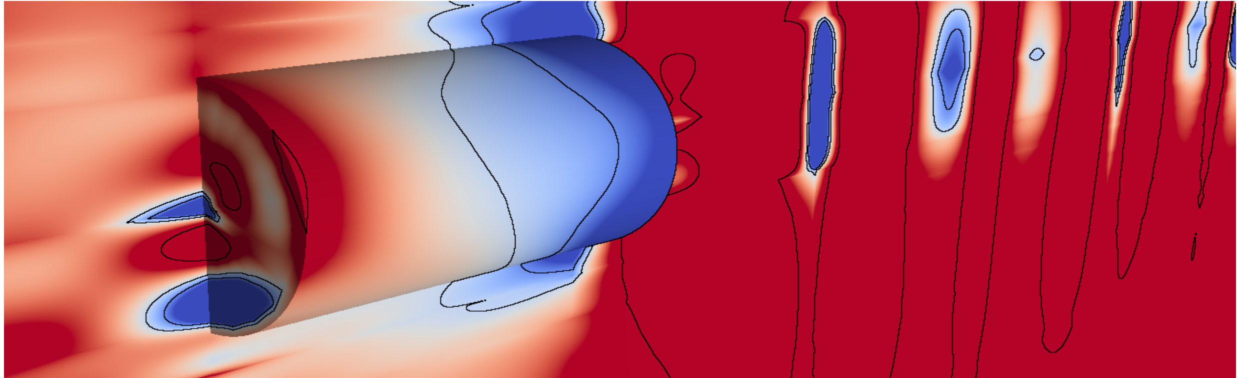


Fig. 1. View of the pressure field of a rotating canister moving through an incompressible fluid. A color map of the field, along with contours of constant pressure, have been applied to the cylinder and the cut-plane.

**Abstract**—We present a GPU-based ray-tracing system for the accurate and interactive visualization of cut-surfaces through 3D simulations of physical processes created from spectral/$hp$ high-order finite element methods. When used by the numerical analyst to debug the solver, the ability for the imagery to precisely reflect the data is critical. In practice, the investigator interactively selects from a palette of visualization tools to construct a scene that can answer a query of the data. This is effective as long as the implicit contract of image quality between the individual and the visualization system is upheld. OpenGL rendering of scientific visualizations has worked remarkably well for exploratory visualization for most solver results. This is due to the consistency between the use of first-order representations in the simulation and the linear assumptions inherent in OpenGL (planar fragments and color-space interpolation). Unfortunately, the contract is broken when the solver discretization is of higher-order. There have been attempts to mitigate this through the use of spatial adaptation and/or texture mapping. These methods do a better job of approximating what the imagery should be but are not exact and tend to be view-dependent. This paper introduces new rendering mechanisms that specifically deal with the kinds of native data generated by high-order finite element solvers. The exploratory visualization tools are reassessed and cast in this system with the focus on image accuracy. This is accomplished in a GPU setting to ensure interactivity.

**Index Terms**—High-order finite elements, spectral/$hp$ elements, cut-plane extraction, GPU-based root-finding, GPU ray-tracing, cut-surface extraction.

## 1 Introduction

Finite element methods are widely used for solving scientific and engineering problems. They are characterized by the discretization of the problem domain into a collection of elements, followed by the construction of an approximate global solution that is specified in terms of a series of local approximations. Many versions of the finite element method use linear interpolation to represent solution values; other versions, such as those considered in this work, represent solutions using higher-order approximating polynomials.

High-order finite element simulations are often visualized using low-order (typically linear) primitives. Performing visualizations in this way is compelling for several reasons: first, there exists an extensive collection of visualization techniques that expect linear primitives as input, and second, modern graphical processing units (GPUs) can render highly complex scenes composed of linear primitives at interactive speeds. While linear approximations can be rendered efficiently, they do not, in general, faithfully represent the high-order data. Methods designed to use the high-order data directly, without the need for an intermediate linear approximation, faithfully represent the data, but are often not interactive.

Linear approximations of high-order data are created by sampling the data at a specified set of points. If the sampling is performed too coarsely, then the approximation will be unable to resolve details in the underlying data, resulting in visualization errors. Conversely, if the sampling is too fine, while the details will be preserved and the visualization may not contain error, it will have used more processing time and other resources than necessary. While there are techniques for addressing the sampling problem [15, 7, 14], it is often desirable to skip the intermediate step and visualize the high-order data directly (i.e., in its native form) in order to avoid the associated approximation errors. By using the high-order data directly, we can know that any features present in the visualization are also present in the data and are not artifacts of the approximation.

The behavior of a high-order scalar field along an arbitrary cut-surface can often be one of the primary questions that must be answered by a simulation. For example, a simulation of a proposed aircraft body may be performed to determine if it will survive the stresses of flying in a variety of scenarios. In such a simulation, users are interested in the behavior of the simulation at specific boundaries, such as

- *B. Nelson and R.M. Kirby are with the School of Computing and the Scientific Computing and Imaging Institute at the University of Utah, E-mail: bnelson,kirby@cs.utah.edu.*
- *R. Haimes is with the Department of Aeronautics and Astronautics, MIT, E-mail: haimes@mit.edu.*

*Manuscript received 31 March 2011; accepted 1 August 2011; posted online 23 October 2011; mailed on 14 October 2011.*
*For information on obtaining reprints of this article, please send email to: tvcg@computer.org.*

the stress field along the aircraft's wing. An effective way to visualize these types of surfaces is through the application of color maps and/or contour lines.

In this work, we describe a new set of methods for rendering color-maps and contour lines on arbitrary cut-surfaces (of which curved-element boundaries are of particular interest), extending the existing methods for applying color-maps to cut-planes [2, 4]. Our system's goal is to generate accurate and interactive images, allowing users to debug the simulation's code, accurately interpret the image, and perform general exploratory visualization. The system uses the high-order data in its native state, without the need for low-order approximation, and uses the the knowledge of the structure and mathematical properties of the underlying fields to provide accurate images.

The remainder of this paper proceeds as follows. In Section 2, we give a brief overview of the state-of-the-art in high-order visualization, paying particular attention to the existing methods for visualizing cut-planes. In Section 3, we give a brief overview of the relevant features of the finite element method, focusing on those features of the high-order field that we use in our algorithm. In Section 4, we discuss the accuracy issues that arise when dealing with high-order data. In Section 5, we show how the system was implemented on a GPU ray-tracing system. In Section 6, we show results of our system, along with timings and other relevant measurements. We conclude in Section 7 with a summary of our findings.

## 2 Related Work

While there are no algorithms that apply color maps to arbitrary cut-surfaces directly, there are several schemes that apply color maps to planar data. In one approach, the color map is generated by what is called a polynomial basis texture [4]. Each basis function used in the high-order field is sampled onto a triangular texture map. The colors in the triangle are not generated by linear interpolation, but instead by the linear combination of the appropriate textures, based on the triangle's order. In this way, a set of basis textures can be generated in a pre-processing step, and then, assuming there is sufficient resolution in the texture, accurate images can be generated for all high-order triangles. Another method uses an OpenGL fragment shader to calculate the field's value at each fragment's location, resulting in more accurate lookup into the color map [2]. Finally, another method analytically calculates the intersection of a plane and quadratic tetrahedra, then uses a ray tracer to apply the color map to the new primitive [21].

Most of the work dealing with the generation of contour lines deals only with 2D high-order elements. A common theme is to generate the contours in an element's reference space (which we will define in Section 3) and then transform them into global (world) space for display. One approach [8] creates contour lines in an element's reference space by subdividing the domain and using linear interpolation within these sub-domains to create a piecewise linear contour. Another approach steps along a direction orthogonal to the field's gradient [1], where each step is controlled by a user-defined step size. A method for generating contour lines over quadrilateral elements by determining the shape of the contour in reference space and then generating a linear polyline to approximate it was developed in [17] and later extended to linear and quadratic triangles in [18].

The only 3D contouring algorithm [6] generates contour lines on cut-planes through finite element volumes. The procedure first locates a seed point for the contour line along the element's boundary. It then steps in a direction orthogonal to the field's gradient, using a user-controlled step size, to generate a polyline representing the contour. It differs from the previously described methods in that the plane is a three-dimensional entity. At each step, the contour can, and often does, move off of the cut-surface. The method introduces a correction term to fix this problem and keep the contour on the cut-plane. As with the other object-space contour methods described, the step size is useful to determine how accurately the polyline represents the contour in world space, but is not as useful in expressing how accurate the final image is, as it can be accurate from one view but have large error in another.

Approaches that use interval methods for generating contours [19] assume that, once a suitable region has been isolated, that the intersection between the implicit and a line segment can be computed efficiently.

Several approaches have been developed for volume rendering high-order fields. An analytic solution to the volume rendering integral was developed in [23] for linear and quadratic tetrahedra. Numerical, point-based solutions for high-order tetrahedra were presented in [24], and solutions for arbitrary elements and order have been developed as well [20]. Approaches for isosurface rendering have been developed for quadratic tetrahedra using analytical calculation of the isosurface in reference space [22] and through ray-tracing approaches [21]. Other approaches include using a ray tracer for arbitrary elements of arbitrary order [11] and a point-based approach that uses particles that actively seek and distribute themselves on the isosurface [9].

In [3] a method for calculating accurate streamlines through high-order fields was developed, and it not only follows the high-order flow more accurately, it also correctly handles the transitions between elemental faces.

## 3 High-Order Finite Elements

A finite element volume is represented by the decomposition of a domain $\Omega$ into a mesh of $n$ smaller, non-overlapping elements $\Omega_i$ such that $\Omega = \bigcup_{i \leq n} \Omega_i$. The four basic element types that are used in this work are the hexahedron, prism, tetrahedron, and pyramid. Each element is associated with three different spaces: world, reference, and tensor. The world space represents the element in its physical position and orientation. In reference space, each element is transformed to a common, element-specific representation. The mapping from the reference space to the world space element is given by the bijection $\chi : \mathbb{R}^3 \rightarrow \mathbb{R}^3$. The tensor space element is the cube $[-1,1]^3$ for all elements, and has a mapping to the reference element $T : \mathbb{R}^3 \rightarrow \mathbb{R}^3$. The mapping between tensor space and world space, which is a bijection *a.e.*, is given by $\Phi = \chi(T(\eta))$. Given that the composition of the mappings is a bijection *a.e.* allows us to convert points between spaces as needed during visualization. An diagrammatic example of these mappings for a tetrahedron is shown in Figure 2.

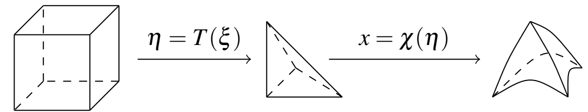



Fig. 2. Illustration of the mapping between tensor (left), reference (middle), and world space (right) for a tetrahedron. Tensor points are denoted by $\eta$, reference points by $\xi$, and world space points by $x$.

For the remainder of this paper, we will use the following notation to help distinguish between points in tensor space and points in world space: points in the world Cartesian space will be indicated by $x$, with individual directions will be specified by $x_1, x_2, x_3$. Similarly, in the local Cartesian space associated with the element's tensor element, points will be specified with $\xi$, and individual components as $\xi_1, \xi_2, \xi_3$.

The solution to a high-order finite element simulation is a polynomial function $F(\xi) \in \mathscr{P}^{N_1,N_2,N_3}$ with respect to the tensor element, where $N_1, N_2, N_3$ denote (possibly) different polynomial orders in the three principle directions. Since $\Phi$ is a bijection *a.e.* (with special care being needed only at a collapsed vertex), we can invert it to calculate the tensor point for a given world point, and then use the tensor point to calculate the field value. While $\Phi^{-1}$ exists for each element, it is, in general, not known analytically and must be calculated numerically when needed. It is also, in general, a non-linear transformation that, when applied to the polynomial field in tensor space, produces a function in world space that, while smooth, is not necessarily a polynomial. The implications for our implementation are that any algorithm that operates in world space cannot assume the field will be a polynomial.

This work deals with Continuous Galerkin (CG) formulations of the finite element method, which require that the fields are continuous across element boundaries, but impose no restrictions on the continuity of any derivatives. Therefore, while the field is $C^p$ on the interior

of each element (with $p \geq 1$ indicating higher levels of smoothness depending on the element's approximating polynomial order and its mapping to world space), the field over the domain $\Omega$ is $C^0$. This assumption of continuity allows us to develop a contouring algorithm, described further in Section 5.3, that is accurate across element boundaries.

## 4 Image Error Characterization

Generating accurate visualizations of high-order finite element volumes is contingent upon identifying and controlling the sources of error in the visualization pipeline. This pipeline takes as input the high-order field as well as a description of the cut-surfaces on which to render the field. The output is a rasterization of the field and cut-surface. A typical implementation, using linear approximations, can contain several different types of error.

The first error is that of approximating the cut-surface. Triangular mesh approximations of cut-surfaces are generally constructed by first sampling the surface at a specified collection of points, then creating a mesh that joins each of these points. When rendering the mesh, the only points that do not contain approximation error are the initial mesh points; all other points in the mesh will be located, at most, some non-zero distance $\varepsilon$ from the true surface. As the number of triangles approximating the surface increases, $\varepsilon$ gets smaller, but it will not, in general, become 0.

If the surface is linear, as in the case of a cut-plane, there will be no error between the approximation and the surface. The error only occurs when the surface is of a higher-order than the approximation.

The second type of error is interpolation error. High-order fields are represented by a polynomial $F(\xi) = \sum_i a_i \phi_i(\xi)$, where $\phi_i$ is the $i^{th}$ basis function. Approaches that approximate the field with linear interpolation introduce error whenever the field is not sampled at nodal points.

Finally, error can be introduced by the process of sampling onto a regular grid of pixels. If the scene contains features that are too fine to be resolved by the pixel spacing, then an error called aliasing can occur.

The focus of this work is on how to minimize triangulation and interpolation errors while producing interactive visualizations. We refer to the combination of these errors as the sampling error. Since rasterization error occurs for all types of visualizations, not just high-order visualizations, we do not address it directly in our analysis. If aliasing error is noticeable in a high-order visualization, it can be addressed through one of the available anti-aliasing techniques.

We define an accurate image of a cut-surface through a high-order data set as one in which each pixel's color is generated based on accurate sampling of the high-order field (i.e., the sampling error is reduced to machine precision). We call this type of image a pixel-exact image. Visualizations based on linear methods are not, in general, pixel-exact, since the triangulation of the cut-surface is not exact and the field between vertices is represented with linear interpolation.

It is possible for linear methods to generate nearly pixel-exact images for particular viewpoints. Assuming that each triangle's vertex is generated by sampling the high-order field, if every pixel in the resulting image contains at least one vertex, then the image will be nearly pixel-exact. The difficulty associated with this approach is that a particular mesh will only generate these images for a subset of all possible viewing parameters and, as we will show in Section 6, it can incur significant costs in terms of computation time required and resource consumption.

## 5 Rendering High-Order Cut-Surfaces

In this section we describe the implementation of our high-order rendering system. While the system supports only cut-surface rendering at the present time, it has been designed to be extensible, which will allow for additional visualization techniques to be added in the future.

The accuracy issues discussed in Section 4 can best be handled by a rendering system that operates on each pixel directly. A ray-tracer is the natural choice in this scenario. Assuming appropriately accurate intersection routines are available, the error associated with locating a point on the cut-surface and calculating the field value is reduced to machine precision. We have implemented high-order ray-tracing systems in the past, and have found them to be too slow to be useful for the types of exploratory visualization we want to perform [11]. Therefore, we have implemented this system on the GPU, leveraging the highly parallel nature of the GPU to process rays simultaneously. Rather than implement our own ray-tracer, we have decided to use the OptiX framework from NVIDIA [13], enabling us to provide an implementation that can run interactively on a typical workstation. It is interesting to note that recent work with volume rendering has been able to reach interactive speeds with a GPU implementation [20].

OptiX provides a number of extension points that allow users to write custom intersection and shading programs. Details can be found in the OptiX programmer's guide [12]. The initial customization point is a ray generation program, which is generally responsible for generating one or more rays per pixel. In our system, the modules described in the following sections are ray generation programs.

Because OptiX is implemented using CUDA, we were able to optimize performance by using strategies common to all CUDA programs, such as avoiding branching where possible and reducing the number of registers required during computation. However, we found it difficult to optimize memory access since access to OptiX memory is provided through an opaque pointer with no guarantees about its location.

An OptiX scene is represented by a graph that controls which geometric objects are visible to a ray, as well as how the ray will traverse the scene. There are two ray-casting scenarios in our system: the initial ray that intersects cut-surfaces, and the secondary rays which are used to find the elements that contain the intersection point. This suggests a scene division where all geometric objects are placed in one graph and the high-order elements in a second. We found that, for performance reasons, the graphs should be shallow. Complicated graphs, such as a graph with a single geometry node for each element in the volume, takes significantly longer when the OptiX engine is initialized, and also adds overhead when launching an OptiX kernel. Since our system is composed of a sequence of OptiX kernels, we found that a poor graph structure can lead to overhead that dominates execution time.

Each node in the graph can be customized by user-defined intersection programs, which are executed to determine if a ray intersects an object, and closest-hit programs, which are executed when the closest intersection point has been determined. The cut-surface intersections described in Section 5.1.1 are implemented as intersection programs associated with the cut-surface nodes. If a ray intersects a cut-surface, the surface's closest-hit program is executed to find the enclosing element and evaluate the field, as described in Sections 5.1.2 and 5.1.3. In the following sections we discuss the OptiX modules that comprise our system.

### 5.1 Primary Ray Module

Rays are represented in parametric form:

$$r = O + tD \tag{1}$$

where $O$ is the ray's origin, $D$ is the ray's direction, and $t$ is the distance along the ray from the origin along the specified direction.

The Primary Ray Module is a ray generation program that first queries the downstream modules to determine the directions of the rays that are needed. It is assumed that a ray through the center of each pixel will be required, but any of the downstream modules can request additional rays. The module then casts rays through each of the requested locations.

The behavior of the ray once it finds an intersection depends on the surface type. This surface-dependent behavior is implemented using an OptiX closest-hit program. If the surface is a cut-surface then a secondary ray is cast to determine which element encloses the intersection point, and the scalar value at the point is then calculated. Otherwise, the surface normal and color are determined. In this way, the scalar values of the field along the cut-surface are calculated and made available to downstream modules, while at the same time, any surface geometry in the simulation can also be rendered.

#### 5.1.1 Ray/Cut-Surface Intersection

Cut-surfaces are generally specified as either implicit 3D surfaces or two-parameter parametric surfaces. Implicit 3D surfaces are defined as the set of all points that satisfy the equation

$$f(x_1, x_2, x_3) = 0 \tag{2}$$

where $f : \mathbb{R}^3 \to \mathbb{R}$. The intersection between an implicit surface and a ray can be found by substituting Equation 1 into Equation 2, yielding:

$$\begin{aligned} g(t) &= f(r_1(t), r_2(t), r_3(t)) \\ &= f(O_1 + tD_1, O_2 + tD_2, O_3 + tD_3) = 0. \end{aligned} \tag{3}$$

Viewed in this way, the intersection test becomes a univariate root-finding problem. For simple implicit functions, this equation can be solved analytically. For example, a cut-plane is a simple implicit function of the form $f(x_1, x_2, x_3) = Ax_1 + Bx_2 + Cx_3 + D = 0$ which, after substitution, becomes a linear equation in $t$ with a trivially obtained solution.

When the implicit surface is more complicated, such as when it is a high-order polynomial, it cannot be solved analytically. In these cases, numeric root-finding techniques are required [5]. In some cases, such as functions representing isosurfaces of high-order fields, the implicit function itself does not have an analytic form, and numeric techniques are required to both evaluate $f$ and find the intersection [11].

The other type of surface of interest in a high-order setting is that of parametric surfaces, defined as $x = p(u, v)$. If $p$ is analytic, it may be possible to convert it into an implicit form, allowing for the direct use of methods already available for implicit forms.

A class of interesting parametric surfaces is that of the faces of the elements themselves. Faces that border simulation geometry are of particular interest because of their relationship to what happens at or near these locations (*e.g.*, pressure at specific locations on an aircraft's wing). Faces are defined as parametric functions:

$$x = \Phi(\xi_i, \xi_j) = \sum_a \sum_b \hat{u}_{ab} \phi_a(\xi_i) \phi_b(\xi_j), -1 \le \xi_1, \xi_2 \le 1 \tag{4}$$

with the ray-face intersection specified as the values for $\xi_1, \xi_2$, and $t$ for which the following holds:

$$\Phi(\xi_i, \xi_j) = r(t). \tag{5}$$

The function $\Phi$ is the mapping function described in Section 3 and does not, in general, have a conversion to a general implicit function.

#### 5.1.2 Point Location

Once the cut-surface intersection point has been found, the next task is to determine the element in which it lies. To find the element, a new, secondary ray is cast from the intersection point in a random direction. This finds the closest element to the intersection point, which is the enclosing element. We constrain our cut-surfaces to lie entirely within the finite element volume; if this were not true, additional testing would be required to determine if the ray/cut-surface intersection point is inside the volume or not.

#### 5.1.3 Field Evaluation

After finding the intersection point $x$ and the element $E$ that contains the point, the field can be evaluated. The field is defined in terms of the local Cartesian coordinate system associated with $E$'s tensor element. As we discussed in Section 3, the mapping between the local tensor space and the global Cartesian coordinate system (in which $x$ exists) is specified by the function $\Phi_e(\xi)$. Because $\Phi_e$ is a bijection *a.e.*, we can obtain the tensor space point for a given world space point by inversion. To calculate the field's value at a world point $x$ inside the element:

$$\hat{F}(x) = F(\Phi_e^{-1}(x)). \tag{6}$$

Unfortunately, $\Phi_e$ does not, in general, have an analytic inverse. We therefore perform the inverse mapping numerically. We use the Newton-Raphson algorithm, which generates the convergent sequence

$$\xi_{i+1} = \xi_i + J^{-1} \left( \Phi(\xi_i) - x \right) \tag{7}$$

where $J$ is the Jacobian of the mapping function given by

$$J_{ij} = \frac{\partial \Phi_i}{\partial \xi_j} \tag{8}$$

with $\Phi_i$ as the mapping in the $i$ direction. This method has well-known stability issues and can fail to converge, even when roots do exist. In our case, since the mapping function $\Phi$ is a bijection *a.e.*, we know that there is a unique real root inside the element, and we are able to position our initial guess close to the root, which greatly increases the probability of finding the root. Our routines do detect and report convergence errors and, in practice, they have not been encountered.

Fields are represented as the tensor product of one-dimensional polynomials:

$$F(\xi_1, \xi_2, \xi_3) = \sum_i \sum_j \sum_k \hat{u}_{ijk} \phi_i(\xi_1) \phi_j(\xi_2) \phi_k(\xi_3). \tag{9}$$

Implemented as written, evaluating the field is an $O(N^4)$ operation, but if we use the sum factorization technique

$$F(\xi_1, \xi_2, \xi_3) = \sum_i \phi_i(\xi_1) \sum_j \phi_j(\xi_2) \sum_k \hat{u}_{ijk} \phi_k(\xi_3) \tag{10}$$

this is reduced to $O(N^2)$. Due to the number of samples that must be taken to provide an accurate visualization, using the sum factorization technique can provide significant performance gains, especially as the field's polynomial order becomes large.

### 5.2 Color Mapping Module

To apply a color map to a cut-surface, we sample the scalar field at the center of each pixel, using the procedures just described, and then use the resulting scalar value as a look-up into the color map. As previously mentioned, this can result in aliasing, for which standard anti-aliasing techniques can be applied.

### 5.3 Contouring Module

A pixel $(i, j)$ belongs to the contour curve for isovalue $\rho$ if it satisfies

$$\exists_{u,v} : P_{ij}(u, v) = \rho \tag{11}$$

where $P$ is the scalar field of the cut-surface projected onto the image plane. As described in Section 3, the field is continuous over the pixel, even if the pixel spans elements. Therefore, we can determine if the isovalue exists in the pixel by finding two points that bracket the isovalue:

$$\exists_{u_a, v_a, u_b, v_b} : P_{i,j}(u_a, v_a) \le \rho \le P_{i,j}(u_b, v_b). \tag{12}$$

Determining if the isovalue exists somewhere in the pixel can be a complicated and time consuming process that requires the determination of the global maximum and minimum scalar value over the pixel. To reduce the complexity of this test, instead of checking the pixel's interior, our algorithm looks for the isovalue along a pixel's edge. This approach is attractive because it detects the same contours as searching the interior detects except for the case where the contour exists entirely in the pixel's interior (see the lower right corner of Figure 3). These types of contours will show up as isolated points in the image and will not help the user interpret the visualization, so the extra processing time to find them need not be taken.

A simple algorithm to determine if a pixel may be part of the contour is to perform point sampling at the four pixel corners. If the values at the corners bracket $\rho$, then the pixel can be marked as part of the contour with no further testing. This method is appealing for several reasons. First, it is fast. Sampling the pixel corners requires $(w+1)(h+1)$ samples, where $w$ is the image width and $h$ the height.

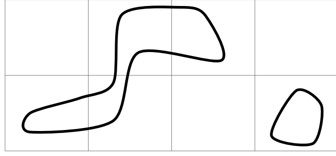


Fig. 3. Contour lines that are not detected by simply checking if the pixel corners bracket the isovalue.

This is only slightly more samples than are required for color mapping, which requires $wh$ samples. Second, if the field is monotonic over the pixel, then this test is also accurate, i.e., it only marks pixels that are part of the contour and doesn't mark pixels that are not part of the contour. Sampling the endpoints of a monotonic function produces the function's range (see Figure 4(a)). While this algorithm cannot guarantee that it will find all contour pixels (because the high-order field is not guaranteed to be monotonic over a pixel's edge), it does find a large percentage of them.

Because the field is not guaranteed to be monotonic, contours can take a variety of forms that will not be detected by the simple corner testing algorithm. As shown in Figure 3, contours can cross edges multiple times and span many pixels and still miss detection by the corner testing algorithm. What we need is a way to obtain an estimate of the range of $f$ using as few samples as possible (since they are expensive and will slow down rendering). An efficient way to do this is through interval arithmetic [10].

Using interval arithmetic, we replace operations on real numbers with operations on intervals. An interval $X$ is defined as

$$X = [\underline{X}, \overline{X}] = \{x \in \mathbb{R} : \underline{X} \leq x \leq \overline{X}\} \tag{13}$$

and, for an arbitrary function $\oplus$ and intervals $X$ and $Y$:

$$X \oplus Y = \{x \oplus y : x \in X \wedge y \in Y\}. \tag{14}$$

The set image of a function $g$ is defined as:

$$g(I) = \{g(x) : x \in X\} \tag{15}$$

and represents the true range of $g$. The interval extension $G$ of $g$ is formed by evaluating the steps to calculate $g$ on interval numbers rather than floating point numbers. The interval extension has the following useful property:

$$G(X) \supseteq g(X) = \{g(x,y,z) : (x,y,z) \in X\}. \tag{16}$$

In other words, if $g$ is evaluated using interval arithmetic on an input interval $X$, the result is a range that is guaranteed to contain the true range of $g$ on that interval.

The interval extension $G$ of $g$ can be very wide, and in some cases can be too wide to be useful. To reduce the width of the interval, we first subdivide the interval into $n$ segments of width $h$, such that the subdivision of $X$ is:

$$X_i = [\underline{X} + ih, \underline{X} + (i+1)h]. \tag{17}$$

We define the interval hull of two intervals as:

$$X \underline{\cup} Y = [min\{\underline{X}, \underline{Y}\}, max\{\overline{X}, \overline{Y}\}]. \tag{18}$$

The interval hull of a subdivided interval is

$$G_n = \underline{\bigcup}_{i \leq n} X_i. \tag{19}$$

Subdividing an interval is useful because of the following property:

$$G(X) \supseteq G_n(X) \supseteq g(X) = \{g(x,y,z) : (x,y,z) \in X\}. \tag{20}$$

In other words, we can decrease the width of an interval computation by subdividing the interval into smaller regions, evaluating each of the sub-intervals separately, then calculating the interval hull of the smaller intervals (see Figure 4).

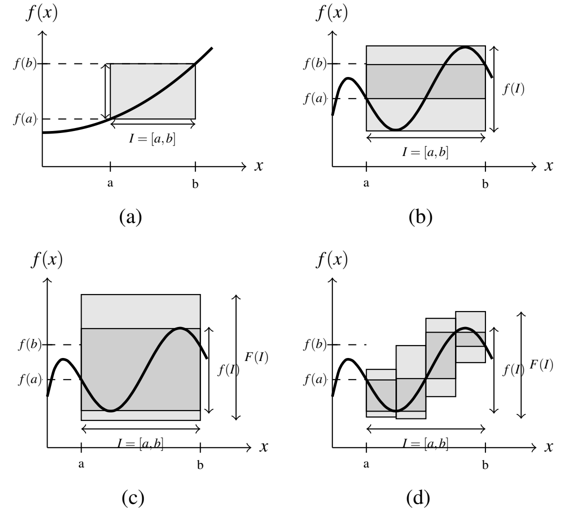



Fig. 4. (a) - If a function $f$ is monotonic on an interval $I = [a,b]$, then the range of $f$ is bounded by $f(I) = [f(a), f(b)]$. (b) - If $f$ is not monotonic, then $f(I) \neq [f(a), f(b)]$. (c) - An interval extension $F$ of $f$ provides bounds that include $f(I)$, i.e., $f(I) \subseteq F(I)$. (d) - Uniform subdivision of the domain produces tighter bounds.

To illustrate how this works, consider the the polynomial $3x^2 + 2x - 4$ which has a range of $[-4.33, 1]$ over $[-1, 1]$. Using the following properties of interval arithmetic:

$$\begin{aligned} X + Y &= [\underline{X} + \underline{Y}, \overline{X} + \overline{Y}] \\ X - Y &= [\underline{X} - \overline{Y}, \overline{X} - \underline{Y}] \end{aligned}$$

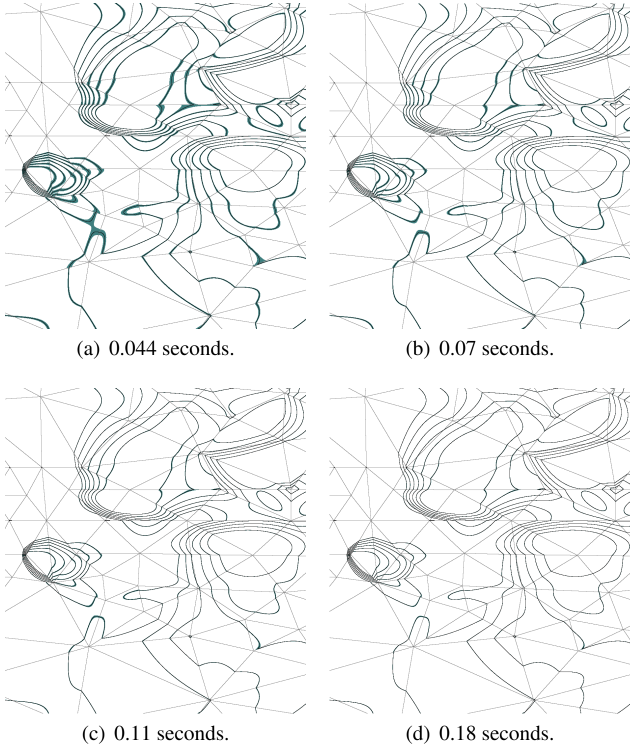
$$X^n = \begin{cases} [\underline{X}^n, \overline{X}^n] & \text{if } \underline{X} > 0 \text{ or } n \text{ is odd,} \\ [\overline{X}^n, \underline{X}^n] & \text{if } \overline{X} < 0 \text{ or } n \text{ is even,} \\ [0, |X|^n] & \text{if } 0 \in X \text{ and } n \text{ is even,} \end{cases}$$

the range can be evaluated as $3[-1,1]^2 + 2[-1,1] - 4 = 3[0,1] + [-2,2] - [4,4] = [-6,1]$. Note the true range, $[-4.33, 1] \subseteq [-6, 1]$. Dividing $[-1, 1]$ into 10 evenly-spaced intervals and evaluating produces a range of $[-4.72, 1]$ and 100 evenly-spaced intervals produces $[-4.37, 1]$.

For high-order fields, calculating the field's range over a pixel's side produces fairly good bounds, as can be seen in Figure 5, where we show that, even without performing any subdivisions, the number of pixels that may contain the isovalue is small. One factor that can contribute to wide interval extensions is the number of operations performed when evaluating an expression. An example of this can be seen in Figure 6, where the number of ambiguous pixels are compared between a $2^{nd}$-order and $6^{th}$-order data set. The $6^{th}$-order data set has more ambiguous pixels than the $2^{nd}$-order data set. Interval arithmetic provides conservative bounds that are obtained without using any derivatives, and are not negatively affected by the presence of extremal features.

The cut-surface contouring algorithm proceeds as follows. First, sample the cut-surface at each pixel's corner. If any two pixel corners bracket an isovalue, mark the pixel. For each unmarked pixel, calculate the approximate bounds of the function along each side. If none of the contours fall within the approximate bound, reject the pixel. Otherwise, mark it as possible. Finally, for each possible pixel, subdivide the edge, taking additional samples as needed, to provide better bounds until a user defined tolerance is met. If contours fall outside the range, reject the pixel, otherwise accept.

Note that, when the algorithm completes, it is possible for some pixels to be marked ambiguously. This cannot be avoided unless the global minimum and maximum over the pixel is calculated. Our system allows for these ambiguous pixels to be colored a different color.



(a) 0.044 seconds. (b) 0.07 seconds. (c) 0.11 seconds. (d) 0.18 seconds.

Fig. 5. Contours generated on a cut-plane of the block/plates data set (see Section 6). Pixels that contain the isovalue are marked in black. Pixels that cannot contain the isovalue are white, and pixels that may contain the isovalue are teal. In 5(a), only corner testing has been performed. In 5(b), one level of subdivision has been performed. Two levels were performed in 5(c), and three in 5(d). We can see that additional testing reduces the number of ambiguous pixels. Rendering times for a 1000×1000 image are given underneath each image. While subdivisions take extra time, the rendering times are still interactive.

The user can then increase the amount of subdivision as needed to reduce the number of ambiguous pixels. In practice, we have found that dividing a pixel's side into eight sub-intervals is enough to handle most ambiguities.

### 5.4 Lighting and Display Modules

The lighting module can be arbitrarily complex. In our version, we implement simple OpenGL style lighting with no shadows.

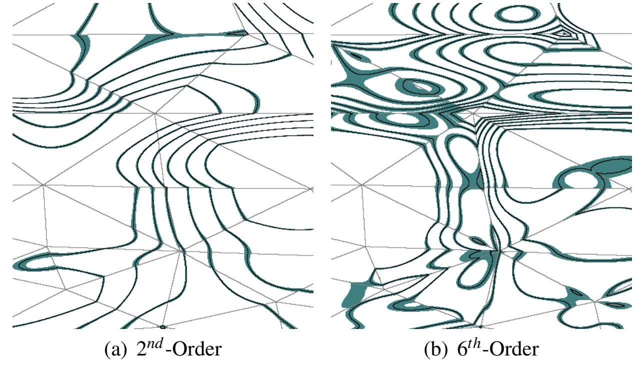
For display, we update the OpenGL color and depth buffers with the values obtained during ray-tracing. Additional OpenGL calls can now be made to render OpenGL data that can co-exist with the ray-traced data.

## 6 RESULTS

The effectiveness of the methods described in this paper are illustrated by applying them to two high-order fluid flow data sets. Note that we will frequently refer to $n^{th}$-order volumes or elements as a shorthand for saying that the field is represented by $n^{th}$-order polynomials, in tensor space, in each direction.

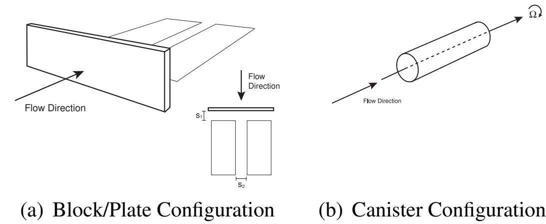
The first example consists of incompressible flow past a block with an array of splitter plates placed downstream of the block. A schematic of this regime is presented in Figure 7(a). As the fluid impinges upon the block, it is diverted around the structure, generating vorticity along the surface. For the purposes of this paper, we will focus our attention on a configuration consisting of a plate spacing of one unit (non-dimensionalized with respect to the block height). The 3D computational mesh consists of $3,360$ hexahedra and $7,644$ prisms. All simulations were performed at a Reynolds Number ($Re$) of 200.

The second data set consists of a rotating canister traveling through an incompressible fluid. A schematic of the flow regime under consideration is presented in Figure 7(b). The 3D mesh consists of $5,040$ hexahedra and 696 prisms, with the computational problem being solved using third-order polynomials within each element. The solutions presented herein were computed at $Re = 1000$ and with an angular velocity of $\Omega = 0.2$.



(a) $2^{nd}$-Order (b) $6^{th}$-Order

Fig. 6. Comparison of ambiguous pixels (teal) between a $2^{nd}$-order (left) and $6^{th}$-order (right) data set.




(a) Block/Plate Configuration (b) Canister Configuration

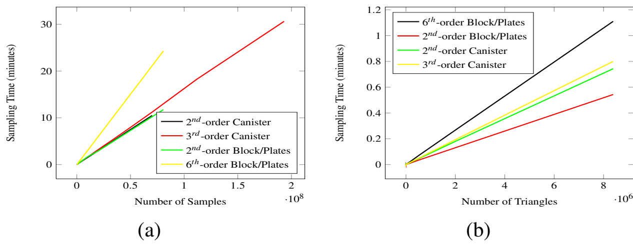
Fig. 7. Schematic showing the basic block/splitter plate (left) and rotating canister (right) configurations under consideration.

All tests described in this section were performed on a desktop workstation equipped with an NVIDIA Tesla C2050 GPU and Intel Xeon W3520 quad-core processor running at 2.6 GHz. All code run on the GPU was implemented in OptiX. Code executed on the CPU was written as single-threaded C++ code. The GPU algorithms were run using 32 bit floating point precision, while the CPU code was run using 64 bit floating point precision. In our tests, there is a negligible performance impact between 32 and 64 bit precision on the CPU, while 64 bit precision on the GPU generally doubles overall execution time. As we will show when we discuss performance, our methods execute at 10-20 frames per second even for large image sizes, so 64 bit precision can be used when necessary without significantly impacting performance.

### 6.1 Linear Comparison Models

Visualizations of high-order data are traditionally performed using linear primitives. Because of this, we will contrast the performance, accuracy, and resource consumption of our methods to those commonly used linear algorithms. We will show that while linear methods can produce acceptable images under the right circumstances, a more effective way to reliably achieve accurate visualizations (under reasonable resource constraints and without intervention) is through the methods described here. The linear test cases were implemented using the Visualization Toolkit (VTK) [16] because it is a well-known system, in current use, and therefore provides a good foundation on which to base our tests.

Two approaches were used for representing the high-order data with the linear structures found in VTK. In the first approach we sampled the entire high-order volume onto a 3D regular grid of points, where the spacing between points is constant. Values between grid points are calculated using linear interpolation. For the second approach a




(a) (b)

Fig. 8. Time to sample a grid of evenly spaced points that span the high-order volume (left), and a triangular mesh representing a cut-plane through the middle of the volume (right).

triangular mesh is created and sampled with the high-order data at the triangle vertices. As with the first volume, data values between vertices are obtained via linear interpolation. Sampling was performed using a CPU-based implementation. We used an octree data structure to accelerate the location of the element enclosing a given sample point. Timings for sampling several high-order volumes onto a grid are found in Figure 8(a), and the timings to sample a cut-plane running through the center of the data set in Figure 8(b).

While sampling onto a 3D grid is the most flexible approach (since so many visualization algorithms can be applied), it is also the most time consuming and quickly becomes prohibitive, especially as the volume order increases. Sampling directly onto the cut-plane, while not as flexible, is fast enough to be practical. The problem with sampling onto a triangular mesh is that the memory required to store the mesh and samples grow roughly in a squared manner (by the spacing). We can produce a cut-plane with $8,388,608$ samples using the $2^{nd}$-order block/plate data set in about 30 seconds, but it consumes 180 MB of memory, compared to 4 MB to represent the entire high-order volume. And, as shown in the examples below, there are not enough samples to produce highly-accurate visualizations.

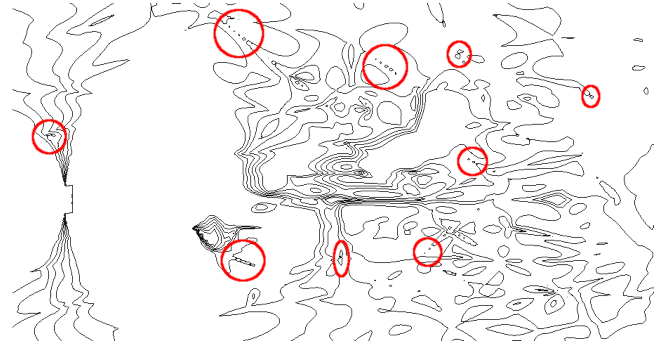
In the remainder of this section, our comparison tests are performed with the triangular mesh-based technique instead of the 3D grid-based approximation. We found that the quality of the grid-based approach is worse (sometimes significantly) than the mesh approach because the cut-surface may span between the grid samples, meaning that the contours and colors produced are interpolated from samples that do not actually lie on the surface. Using the triangular mesh, however, guarantees that all samples are on the cut-surface, providing more accurate images and better comparisons.

### 6.2 Contouring

While there are contouring algorithms available for high-order methods (as discussed in Section 2), we are unable to provide direct comparisons with those methods because they either are restricted by the type of element or the maximum order. Contours for the linear data were produced by the vtkContourFilter. The results are shown in Figure 9.

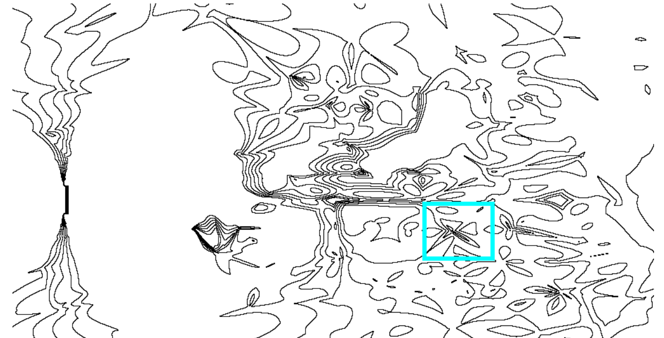
The linear approximation in Figure 9(b) is a significant improvement when compared to the more coarsely sampled volume, and it is difficult to see much difference between this image and our high-order method. However, this is only true at the current resolution, and if we zoom into the image, we can see that there are still significant errors present. While further sampling can improve the generated contour, there are limits to the sampling resolution (due to the amount of resources consumed). And importantly, the method described in this paper performs at about the same speed as the coarsely sampled image.

In the coarsely sampled contour image (Figure 9(a)), there are many errors which have been noted with a red circle: incorrect topology, missed contours, and incorrect shapes. What is interesting is that it is not possible to determine if these contours are accurate from the image alone. We need either a cut-plane with greater resolution or an image from our system to notice the inaccuracy.

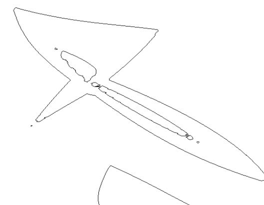


(a) $524,288$ Triangles. VTK Contour Generation Time = 0.265 seconds.

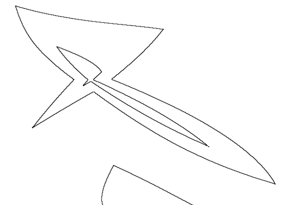
(b) $8,388,608$ Triangles. VTK Contour Generation Time = 3.5 seconds.



(c) High-Order Rendering. Rendering time for $2000\times2000$ image = 0.3 seconds.



(d) Detail View of 9(b)

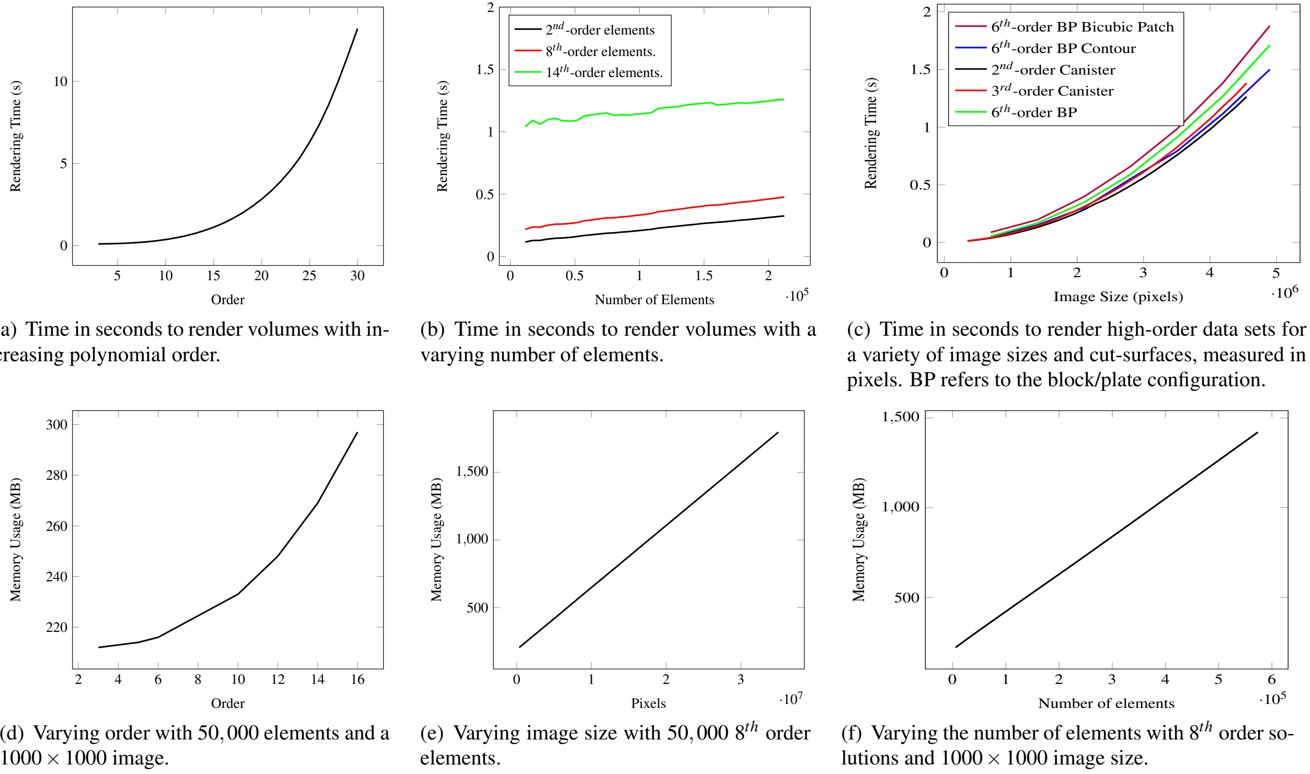


(e) Detail View of 9(c)

Fig. 9. Comparison between pressure contours of the block/plane data set generated using linear methods (9(a) and 9(b)) and our high-order method (9(c)). The approximation in 9(a) has several significant errors, marked in red, which disappear when using a smaller sampling. The highlighted area in 9(c) is shown in more detail in 9(d) for linear interpolation and 9(e) for our system. The contours generated by VTK, which use the same amount of time as our system, have considerably more error.

### 6.3 ColorMaps

Figure 11 contrasts a comparison between linear approximations of a cut-plane through the canister data set with the renderer described in Section 5. Since the cut-plane is an inherently linear cut-surface, there is no surface approximation error, so differences between our method




(a) Time in seconds to render volumes with increasing polynomial order.

(b) Time in seconds to render volumes with a varying number of elements.

(c) Time in seconds to render high-order data sets for a variety of image sizes and cut-surfaces, measured in pixels. BP refers to the block/plate configuration.

(d) Varying order with 50,000 elements and a $1000 \times 1000$ image.

(e) Varying image size with 50,000 $8^{th}$ order elements.

(f) Varying the number of elements with $8^{th}$ order solutions and $1000 \times 1000$ image size.

Fig. 10. Performance results and memory usage for rendering high-order data using our system. For typical usage scenarios, the parameter that impacts performance the most is image size.

and the linear methods are due solely to the differences between linear and high-order interpolation.

As expected, the images get progressively closer to the image generated by the techniques described in Section 5 as the number of samples increase. However, even at the very fine sampling (shown here), there are still subtle errors. Additionally, the memory required to store the mesh is large which requires a lengthy pre-preprocessing step. It is interesting to note that, unlike the contour images, it can be visually obvious when a color map needs to be further refined by the presence of sharp and "boxy" gradients.

### 6.4 Performance

For the methods described in this paper to be useful, they must not only be accurate but interactive as well. For each test, we rendered a view of a cut-plane and cut-cylinder where the entire image was covered. Since the rendering pipeline is restarted from scratch with every view change, there was no need to run the timing tests with a variety of viewing changes. We executed the tests 100 times and reported the average time to render the entire scene.

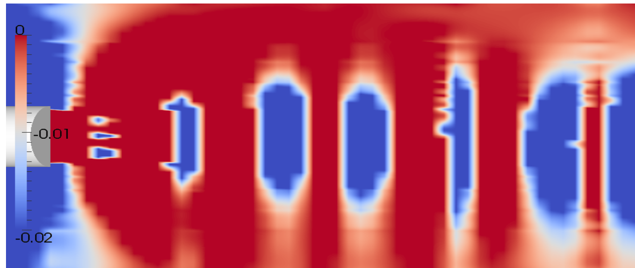
Since we did not have fluid flow simulation data higher than $6^{th}$ order, synthetic data sets were generated to obtain the timings. We were interested in how timing was related to the simulation's order, number of elements, and the overall image size. Timing results are shown in Figure 10 and, unless specifically noted otherwise, are valid for both the contouring and color mapping module (as they generally take the same amount of time). In Figure 10(a), it can be seen how increasing the field's order impacts performance. If the order is increased significantly, then it will impact performance negatively, but for typically-used orders, around 2-14, rendering times are not significantly impacted. In Figure 10(b), we tested our system on large data sets with up to 200,000 elements and it can be seen that our system's speed is not strongly related to the number of elements. In Figure 10(c), we can see that the most important factor governing the performance of our system is the image size. This figure also illustrates the impact of using more complicated cut-surfaces, specifically bicubic patches. While overall rendering time is increased with more complicated surfaces, it is still interactive. This has several positive implications. First, if the system is not as interactive as desired, additional speed can be gained by using a smaller viewport. Second, the system is capable of handling large high-order data sets interactively, and there is no indication of an upper limit to the number of elements that can be supported (except for the size of the native solution that must fit in the GPUs memory).

In Figures 10(d), 10(f), and 10(e), we show the GPU memory required by our system. We start with a 5,000 element, $8^{th}$ order volume rendered in a $1000 \times 1000$ image, then vary the number of elements, polynomial order, and image size. These graphs show that our system is capable of storing large volumes of high order on a single GPU. Increasing the image size results in a linear growth in memory usage due to the constant amount of per-pixel memory required for calculation and rendering. Increasing the number of elements also scales linearly, as the number of coefficients per element is constant for a given polynomial order. When increasing the polynomial order, the number of coefficients required to support the solution grows as $O(n^3)$, where $n$ is the polynomial order, leading to the memory growth shown in Figure 10(d).

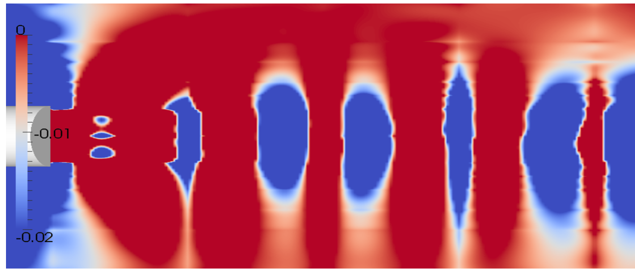
## 7 Conclusion

Our system was motivated by the lack of existing visualization techniques capable of interactively and accurately rendering color-maps and contour lines on arbitrary cut-surfaces. In this paper, we have described a new system that is capable of rendering these surfaces interactively and accurately while using the high-order data in its native form (i.e., we do not need to resample onto lower-order constructs). We have also shown that the most important factor for determining rendering speed is the size of the final image, indicating that our system can efficiently handle high-order data sets with a large number of elements with a large number of modes. An additional benefit is that these interactive frame rates are achievable on commodity GPUs, meaning simulation scientists can easily perform even the most demanding visualizations at their workstations.

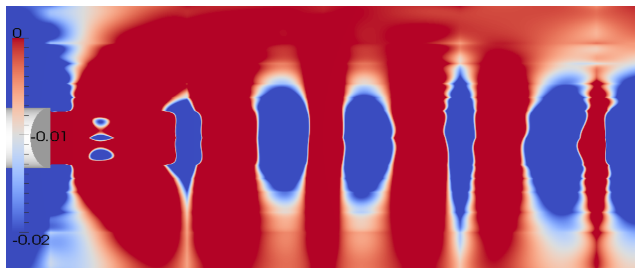
While our system is useful in its current form, additional function-




(a) Cut-plane with 902,289 triangles, VTK Rendering Time = 0.08 seconds.




(b) Cut-plane with 8,388,608 triangles, VTK Rendering Time = 2.0 seconds.




(c) Pixel-exact cut-plane color map. Rendering time is 0.015 seconds for a 1800×800 image.

Fig. 11. Color maps for a cut-plane through the canister data set with a coarse sampling (11(a)), a fine sampling (11(b)), and pixel-exact using our method (11(c)).

ality, such as isosurface generation and volume rendering, is required before it can become a general-purpose high-order visualization system. We have an algorithm for isosurface generation in arbitrary high-order volumes [11], but it is currently implemented as a CPU-based algorithm and, while accurate, is far from interactive. Our current system already implements several features that are required by this algorithm, namely, sampling the field at arbitrary points and performing ray-element intersections. The additional work we anticipate is the creation of function projection and root finding modules. Moving the CPU specific code to a GPU setting is not sufficient to guarantee interactivity, as the isosurface module requires significantly more samples than cut-surface rendering, and iterative root finding can be challenging on a GPU.

## Acknowledgments

This work is supported under ARO W911NF-08-1-0517 (Program Manager Dr. Mike Coyle) and Department of Energy (DOE NET DE-EE0004449). Infrastructure support provided through NSF-IIS-0751152.

## References

[1] J. Akin, W. Gray, and Q. Zhang. Colouring isoparametric contours. *Engineering Computations*, 1:36–41, 1984.

[2] M. Brasher and R. Haimes. Rendering planar cuts through quadratic and cubic finite elements. In *Proceedings of the conference on Visualization '04*, VIS '04, pages 409–416, Washington, DC, USA, 2004. IEEE Computer Society.

[3] G. Coppola, S. J. Sherwin, and J. Peiró. Nonlinear particle tracking for high-order elements. *J. Comput. Phys.*, 172:356–386, September 2001.

[4] B. Haasdonk, M. Ohlberger, M. Rumpf, A. Schmidt, and K. G. Siebert. Multiresolution visualization of higher order adaptive finite element simulations. *Computing*, 70:181–204, July 2003.

[5] A. Knoll, Y. Hijazi, I. Wald, C. Hansen, and H. Hagen. Interactive ray tracing of arbitrary implicits with simd interval arithmetic. In *Proceedings of the 2nd IEEE/EG Symposium on Interactive Ray Tracing*, pages 11–18, 2007.

[6] G. D. Kontopidis and D. E. Limbert. A predictor-corrector contouring algorithm for isoparametric 3d elements. *International Journal for Numerical Methods in Engineering*, 19(7):995–1004, 1983.

[7] A. O. Leone, R. Scateni, S. Pedinotti, L. Marzano, P. Marzano, E. Gobbetti, E. Gobbetti, and S. S. Pedinotti. Discontinuous finite element visualization. In *8th International Symposium on Flow Visualisation*, 1998.

[8] J. L. Meek and G. Beer. Contour plotting of data using isoparametric element representation. *International Journal for Numerical Methods in Engineering*, 10(4):954–957, 1976.

[9] M. Meyer, B. Nelson, R. Kirby, and R. Whitaker. Particle systems for efficient and accurate high-order finite element visualization. *IEEE Transactions on Visualization and Computer Graphics*, 13:1015–1026, 2007.

[10] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.

[11] B. Nelson and R. M. Kirby. Ray-tracing polymorphic multidomain spectral/hp elements for isosurface rendering. *IEEE Transactions on Visualization and Computer Graphics*, 12:114–125, January 2006.

[12] NVIDIA. *NVIDIA OptiX Ray Tracing Engine Programming Guide*, version 2.1 edition, 03 2011.

[13] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. Optix: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010.

[14] J.-F. Remacle, N. Chevaugeon, É. Marchandise, and C. Geuzaine. Efficient visualization of high-order finite elements. *International Journal for Numerical Methods in Engineering*, 69(4):750–771, 2007.

[15] W. Schroeder, F. Bertel, M. Malaterre, D. Thompson, P. Pebay, R. O'Barall, and S. Tendulkar. Framework for visualizing higher-order basis functions. In *Visualization, 2005. VIS 05. IEEE*, pages 43 – 50, 2005.

[16] W. Schroeder, K. M. Martin, and W. E. Lorensen. *The visualization toolkit (2nd ed.): an object-oriented approach to 3D graphics*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[17] C. Singh and D. Sarkar. A simple and fast algorithm for the plotting of contours using quadrilateral meshes. *Finite Elem. Anal. Des.*, 7:217–228, December 1990.

[18] C. Singh and J. Singh. Accurate contour plotting using 6-node triangular elements in 2d. *Finite Elem. Anal. Des.*, 45:81–93, January 2009.

[19] J. M. Snyder. Interval analysis for computer graphics. *SIGGRAPH Comput. Graph.*, 26:121–130, July 1992.

[20] M. Üffinger, S. Frey, and T. Ertl. Interactive high-quality visualization of higher-order finite elements. *Computer Graphics Forum*, 29(2):337–346, 2010.

[21] D. F. Wiley, H. Childs, B. Hamann, and K. Joy. Ray casting curved-quadratic elements. In O. Deussen, C. D. Hansen, D. Keim, and D. Saupe, editors, *Data Visualization 2004*, pages 201–209. Eurographics/IEEE TCVG, ACM Siggraph, 2004.

[22] D. F. Wiley, H. R. Childs, B. F. Gregorski, B. Hamann, and K. I. Joy. Contouring curved quadratic elements. In *Proceedings of the symposium on Data visualisation 2003*, VISSYM '03, pages 167–176, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[23] P. L. Williams, N. L. Max, and C. M. Stein. A high accuracy volume renderer for unstructured data. *IEEE Transactions on Visualization and Computer Graphics*, 4:37–54, January 1998.

[24] Y. Zhou and M. Garland. Interactive point-based rendering of higher-order tetrahedral data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):2006, 2006.