Final Report
Automated Extraction of Flow Features
NASA Marshall Space Flight Center Grant #NAG8-1872

Robert Haimes
Department of Aeronautics & Astronautics
Massachusetts Institute of Technology
haimes@mit.edu

June 10, 2005

1 Introduction

Computational Fluid Dynamics (CFD) simulations are routinely performed as part of the design process of most fluid handling devices. In order to efficiently and effectively use the results of a CFD simulation, visualization tools are often used. These tools are used in all stages of the CFD simulation including pre-processing, interim-processing, and post-processing, to interpret the results. Each of these stages requires visualization tools that allow one to examine the geometry of the device, as well as the partial or final results of the simulation. An engineer will typically generate a series of contour and vector plots to better understand the physics of how the fluid is interacting with the physical device. Of particular interest are detecting features such as shocks, re-circulation zones, and vortices (which will highlight areas of stress and loss). As the demand for CFD analyses continues to increase the need for automated feature extraction capabilities has become vital.

In the past, feature extraction and identification were interesting concepts, but not required in understanding the physics of a steady flow field. This is because the results of the more traditional tools like; iso-surface, cuts and streamlines, were more interactive and easily abstracted so they could be represented to the investigator. These tools worked and properly conveyed the collected information at the expense of a great deal of interaction. For unsteady flow-fields, the investigator does not have the luxury of spending time scanning only one "snap-shot" of the simulation. Automated assistance is required in pointing out areas of potential interest contained within the flow. This must not require a heavy compute burden (the visualization should not significantly slow down the solution procedure for co-processing environments). Methods must be developed to abstract the feature of interest and display it in a manner that physically makes sense.

2 A Secondary Flow Surface

2.1 Definition

The concept of secondary flow in turbomachinery is not well defined, but commonly referenced. Some attempts at rigorously defining this idea include:

- ...component of absolute vorticity in the direction of the relative streamline [Hawthorne, 1974]
- Secondary flow in broad terms means flow at right angles to the intended primary flow [Cumpsty, 1989]
- Due to viscous effects, endwalls divert primary flow produced by blades and vanes, to give rise to what has come to be called secondary flow [Bradshaw, 1996]

Of the three definitions listed above only [Cumpsty, 1989] provides a definition that could be made operational. What is required is the notion of primary flow, which we can define. Unfortunately by the time we get a CFD solution the notion of "intended" is lost.

The desire to view perturbations from the *primary* flow direction can give insight into the viscous, reverse flow and vortical effects that deviate from the design. To this end it is obviously desirable to be able to generate 2D vector plots that display the secondary flow given a traditional CFD solution.

Secondary flow plots are usually displayed in a passage between blades or just downstream from the trailing edge. The arrows are generated from a frame of reference that is relative to the passage in question (i.e. absolute for fixed rows and moving for rotors). This obviously points to a difficulty in areas between stators and rotors: what is the appropriate frame of reference?

2.2 Algorithm

It would clearly be desirable to have a scheme that could maximize the primary flow through a constructed surface. This could be done by defining a *pivot* point in the channel that reflects the some centroid of the passage or flow. A surface that goes through the point can then be generated. By adjusting the position of this surface the *best* fit can be found. This surface can then be used to view the secondary flow by projecting the vector field data onto the surface.

2.2.1 Primary Flow definition

The goal here is to calculate the mass-averaged quantities in the channel. This should be done on a grid plane or a cut through the passage that is orientated so that all bounds of the cut are walls (if possible). The following can be done with either a plane (all surface facets have the same normal n) or an analytical surface where the normal for cut facets can change.

Compute surface integrals:

$$\begin{aligned}
\bar{A} &\equiv \iint \vec{n} dA \\
\bar{M} &\equiv \iint [\rho\vec{q}\cdot\vec{n}]\, dA \\
q_0 &\equiv \bar{M}/\bar{A} \\
\vec{X}_0 &\equiv \iint \left[\vec{X}\rho\vec{q}\cdot\vec{n}\right] dA/q_0
\end{aligned}$$

where $\bar{M}$ is the mass-averaged flux, q_0 is the mean velocity and $\vec{X}_0$ is the mass-averaged center of the flow.

2.2.2 Newton-like Iteration to Maximize Primary Flow

By selecting various cuts that pass through $\vec{X}_0$ we can adjust the normal (in the case of a simple plane) in an iterative loop so that we maximize $q_{z'}$ (the velocity perpendicular to the plane):

$$\vec{n} = |q_0|$$

Note that new surface integrals are recomputed during each iteration. This will also change the position $\vec{X}_0$.

Using a planar cut this technique takes about 3 to 4 iterations to converge (i.e. the normals returned do not differ by some small factor). This Newton-like convergence is most always seen unless the planar cut is adjusted so that a new portion of the flow field is exposed.

When converged, this provides a view of the data that displays secondary flow when the normal velocity component is removed.

2.3 Discussion

In practice this algorithm works well but did require a number of operational adjustments. These included:

2.3.1 Passage of Interest

The fast cut algorithms are based on *Marching Cubes* and are performed on a 3D element at a time. The result is a set of disjoint polygons that reflect the portion of the surface that cuts through the cell. The notion of where in the domain the fragments come from is usually lost. So if the simulation contains more than a single passage the cut data can easily contain fragments from elsewhere in the simulation. This will corrupt the primary flow calculation in that we are no longer focused on a single passage.

The solution is to reconnect the fragments into complete (and bounded) surfaces. Once this is done a seed point can be located within the bounded surfaces so that one can be selected. Only those polygons that are within the selected region are used in the calculations.

The cut algorithm used constructs the surface in a Finite-Element sense (that is, a list of nodes that reflect the 3D edges being cut is constructed and the polygons refer to indices in that list). The reconnection is performed via a polygon side-matching algorithm based on the indices (not floating-point locations). This is unique and robust. Any side that is seen by two polygons is interior to the region. A side with only a single polygon is bounding the region.

2.3.2 Multi-block simulations

In multi-block simulations the volumes represented by the blocks can abut or overlap. The individual cell definitions are usually block specific so that even if the blocks maintain a larger contiguous volume, it is usually not apparent by the time one looks at the fragments from the *Marching Cubes* results. When reconnecting the regions the results will reflect the block boundaries and not the actual bounds of the cut. The regions need to be placed back together.

When performing streamlining, it is traditional to use the "IBlank" data to inform the software how the blocks are connected. When one pierces a cell on a face where the "IBlank" data indicates that a *jump* to another block is required, the "IBlank" index contains the accepting block. Initially, this data was used to attempt to flood the region from the target surface fragments to connecting blocks. This was found to be unreliable.

A much more expensive technique was developed. This involved producing a bounding-box around each region as a first step. All regions (that have not been included) and have bounding boxes that overlap the start region are examined. Each point on the exterior of the start region is compared to all fragments of the candidate regions. If it is found that any point is interior, then the new region is considered part of the calculation and this process is then recursively applied where this candidate becomes the start region.

In this way the seed point fills all connecting and overlapping areas and the calculation can proceed on that "passage".

2.3.3 Tip leakage simulations

When performing the secondary flow algorithm on a simulation that displays tip leakage there is a natural connection between passages. With the algorithm described above there will be flooding into other passages. This will corrupt the primary flow calculation.

This problem has been taken care of if the simulation is multi-block and there are individual blocks that represent the tip flow regions. The flooding can be "dammed" by informing the technique not to use certain blocks as candidates.

2.3.4 Frame of reference

In multi-stage calculations care needs to be taken so that algorithm sees data in a consistent frame of reference. This means that when looking at the secondary flow in a rotor, all velocity field values should be in the rotating frame. It is important that the data in the stators be transformed so that the technique does not see any discontinuities in the velocity field.

This then means that if one were to traverse the machine from upstream to down that there will be a number of changes of reference. These should be done while the resultant planar cut is in the zone between blades.

2.4 API addition to FX

The **F**eature e**X**traction toolkit **FX** contains an infrastructure that can handle the various different methods that a CFD solution can be discretized. This toolkit, unlike most visualization systems, is lightweight because no drawing and/or GUI functions are supported. In general, the input is the CFD solution and output is various forms of geometry.

2.4.1 FX_MeanFlow

FX_MEANFLOW(XPOS, VNORM, DAM)

This subroutine given the start position and plane normal computes the mass-averaged "center" of flow and the mass averaged velocity.

float XPOS[3]	On input the position that sets the plane given the normal VNORM. On output, the mass averaged position on the planar cut is returned.
float VNORM[3]	On input the normal that sets the family of planes to use to produce the cut. On output VNORM is filled with the mass averaged velocity through the cut.
int *DAM	Pointer to the status of each block (for multi-block cases) to act as a "dam" for the flooding procedure. Zero indicates that flooding through the block is OK, a one is the flag to NOT use this block. NOTE: may be NULL to specify no "damming".

2.5 Status

A paper was written and presented on this topic at the AIAA Aerospace Sciences Conference and Exhibition in January of 2005. The paper is appended to the end of this report.

This feature technique has also been passed on to General Electric. The people responsible for the visualization codes that both GE Aircraft Engines and GE Power Generation use are located at GE Global Research. Stuart Connell manages this effort and he has incorporated the Secondary Flow finder into NPLOT3D (their visualization "workhorse" code). The continual feedback is that this feature is useful.

3 Field Interpolation

There was an effort at MSFC to be able to accurately interpolate the data from one mesh onto another where the nodal positions do not match. This interpolation can be performed in a number of ways (this is due to the fact that finite volume/finite difference CFD does not actually define a cell-based interpolant). If the interpolation is done without some accuracy, then the solution on the target mesh may be far from converged (even if the source solution was converged and the geometries are the same). This situation becomes worse in meshes dominated by boundary layer stretching – errors in these regions are easy to generate and have a significant effect.

The interpolation routines used for streamlining and unsteady particle tracing were applied to this mesh-to-mesh problem with great success.

4 Rendering of Higher-Order Finite Elements

Numerical methods are widely used throughout academia and industry to solve physical problems when experimental data is difficult to obtain. The details of these methods can vary greatly, but they all essentially solve a set of governing equations by discretizing the domain of interest and solving an analogous formulation at the discrete points or nodes. Once a solution has been generated for these nodes, then data over the entire domain can be obtained by interpolation. The simplest way to interpolate is to assume linearity within each cell based on the vertices that support that element. There are a number of ways available to then view this data, since most visualization techniques are based on the assumption of linear interpolation. However, there are many situations in which it is advantageous to solve the discrete equations using a non-linear basis or higher order elements. This can mean using anything from the standard polynomial Lagrange basis to a scheme as complicated as a hierarchical basis or spectral elements. One obvious difficulty with using higher order numerical methods is that there is no simple way to visualize the data in its native form (since most current visualization software uses a linear basis). This renders higher order methods much less useful. Understanding of numerical results and new insight is often only possible when one can accurately visualize the massive amounts of data produced.

Accurate rendering of nonlinear data cannot be performed efficiently using only the standard OpenGL API, since all OpenGL primitives are inherently linear. Higher order data can be interpolated and rendered quite simply and quickly by utilizing the flexibility of modern graphical processing units (GPUs). In addition to rendering surfaces, one important technique used in scientific visualization is the generation planar cuts through 3D field data. This can be accomplished through a combination of selective refinement of the elements and accessing the programmable shaders inside the GPU.

4.1 Discontinuous Finite Elements

One popular group of numerical techniques, the Finite Element Methods (FEM), are particularly convenient when dealing with complex geometries or unstructured computational meshes. The FEM simplifies the solution scheme by mapping every element in the mesh to a master *reference* element, and then scalar interpolation can be performed using shape functions as a basis.

When rendering continuous data, neighboring elements share both the location and field data of common nodes. The use of collected primitives (polytriangles, quad meshes and etc.) can speed up the display time since the support data needs to be passed along the graphics pipeline fewer times. However, the direct goal of this research was to visualize flow solutions generated using the Discontinuous Galerkin (DG) Method. As such, any scheme developed should be able to naturally handle discontinuities (at element faces) in the scalar fields being visualized. The simplest way to accomplish this is for each element to independently store data for all of its basis nodes. Even though the physical location of shared nodes is the same between neighboring elements, nodes must be respecified for each element in which they appear. The goal is to have a method that allows for easy handling of both continuous and discontinuous data with the acknowledgement that there will be some lose of the speed benefits in comparison to the use of collected primitives for continuous data.

4.2 Visualization Tools

The status of the implementation of commonly used visualization tools for higher-order elements is listed below.

4.2.1 Surface Rendering

The coloring (and lighting) of the surface patches is done in an accurate manner. What is not properly handled, at this point, are curved triangles. OpenGL only rasterizes planar fragments, therefore in order to precisely render curved patches, a method to cover the *shadow* of the patch is required. This geometric fragment is view dependent and therefore changes as the view matrix is adjusted. This portion of the algorithm has not been completed.

What has been accomplished is that a p_1, p_2 and p_3 scalar evaluators have been implemented. Unlike OpenGL where interpolation is performed in color space, here proper scalar interpolation is computed in the graphics hardware and the color applied from a colormap stored in texture memory. Once the color has been found, the same interpolation can be performed on the geometry. This can give an accurate normal on the curved patch. This normal is the one that then gets applied for the lighting calculations. Also, the depth is properly adjusted (and not taken from the linear interpolation of the fragment). This does a remarkably good job in providing a visual representation of the patch even though it is based on the linear raster positions.

4.2.2 Planar Cuts

A scheme to properly render cutplanes has been implemented. Please see the attached paper *Rendering Planar Cuts Through Quadratic and Cubic Finite Elements.* This paper has been published and presented at the IEEE Visualization conference in October 2004.

4.2.3 Iso-surfaces

Preliminary efforts have begun. The algorithms to render each type of intersection for linear elements are the same as with planar cuts. The crucial difference is that iso-surface will, in almost all cases, be guaranteed not to be planar.

However, it may be possible to render the isosurface with scalar value, s^*, by bounding it with linear primitives. Based on screen position, x_s, of each pixel on the bounding shadow, the depth is adjusted until the point on the isosurface, x, is found such that x_s lies on top of x (i.e. x and x_s have the same screen coordinates but different depths). To find x, $|s - s^*|$ is first minimized by performing a search of points inside the element that lie beneath x_s, then the fragment can rejected or drawn based on whether or not $s = s^*$. Performing this search would be relatively expensive, so acceptable values of s will lie close to s^* within some bounds set by the accuracy of the search. Under some viewing transforms, the isosurface can curve behind itself, which means there can be multiple solutions, x, that all lie on top of x_s. In this case, the several solutions should be compared using the depth test to determine which one is displayed.

How are the bounding shadow primitives determined to render the isosurface? The faces of the congruent tetrahedron used to generate the cutplane shadow would certainly cover the isosurface intersection, since it captures the entire element by design. But using those triangles could produce many extraneous fragments.

4.2.4 Steamlines

This has not been started. For continuous data higher order interpolation is not a problem. The normal streamline and unsteady particle tracer is only a function of the velocity field (at optionally its gradient) at requested points. FEM is designed to provide accurate interpolation. Routines are required for each type of element supported in the simulation.

There is a problem for DG simulations. Many of the numerical techniques used for particle/streamline integration assume continuous field data. It is not clear what will happen to the results when there are jumps seen at element boundaries. Those techniques like variable step Runge-Kutta integration will fail. In fact, the concept of a streamline in a discontinuous simulation may not be well defined.

4.3 Status

The student performing much of this effort, Michael Brasher, graduated with his Masters degree in August 2004.

Automated Extraction of Secondary Flow Features

Suzanne M Dorney*
NASA Marshall Space Flight Center, MSFC, AL 35812
suzanne.m.dorney@nasa.gov

Robert Haimes+
Department of Aeronautics and Astronautics
Massachusetts Institute of Technology, Cambridge, MA 02139
haimes@mit.edu

ABSTRACT

The use of Computational Fluid Dynamics (CFD) has become standard practice in the design and development of the major components used for air and space propulsion. To aid in the post-processing and analysis of CFD results, many researchers now use automated feature extraction utilities. These tools can be applied in order to detect the existence of such features as shocks, vortex cores and separation/re-attachment lines. The existence of secondary flow is another feature of significant importance to CFD engineers because it highlights regions of increased losses. Although secondary flow is relatively well understood there is no commonly accepted mathematical description of it. This paper will present a definition for secondary flow and one approach for automatic detection and visualization of this feature.

INTRODUCTION

The use of Computational Fluid Dynamics (CFD) has become standard practice in the design and development of the major components used for air and space propulsion. Many of today's advanced computer simulations create datasets containing as many as a billion pieces of information for a single steady-state run. Clearly, transient simulations of the same spatial fidelity stress the available computer resources. The sheer size of this data results in an exceedingly difficult and time-consuming analysis process.

The task of interrogation and interpretation of this voluminous information is required so that the knowledge contained within the simulation can be extracted. The problem is becoming more significant as improvements in computational performance result in these large-scale simulations becoming more commonplace. Today, computational performance is increasing an order of magnitude every 3.5 years. Simulations once used only for exploration are now available during design and parametric studies.

Traditional interactive visualization is used to probe the data in order to locate and identify physical phenomena, or to identify limitations in the simulation process. However, as the frequency of the large-scale simulations increases in the design process, new approaches must be developed to enable the design engineer to process the information in a timely fashion. Specifically there needs to be closer integration of the traditional analysis stages (pre-processing, solver and post-processing). One scenario is to employ visualization throughout the simulation process.

* Computer Scientist.
+ Principal Research Engineer.

Fluid flow features such as vortices, separation, boundary layers and shocks are items of interest that can be found in the results obtained from CFD simulations. Most visualization systems provide users with a suite of general-purpose tools (e.g., streamlines, iso-surfaces, and cutting planes) with which to analyze their datasets. In order to find important flow features, users must interactively explore their data using one or more of these tools. Scientists and engineers that use them on a regular basis have reported the following drawbacks:

- Exploration Time -- Interactive exploration of large-scale CFD datasets is laborious and consumes hours or days of the scientists/engineers time.

- Field Coverage -- Interactive visualization techniques produce output based on local sample points in the grid or solution data. Important features may be missed if the user does not exhaustively search the dataset.

- Non-specific -- Interactive techniques usually reveal the flow behavior in the neighborhood of a flow feature rather than displaying the feature itself.

- Visual Clutter -- After generating only a small number of visualization objects (e.g., streamlines, cutting planes, or iso-surfaces) the display becomes cluttered and makes visual interpretation difficult.

It is clear that these tools do not directly answer the questions of the CFD investigator. An expert is required to infer the underlying fluid flow phenomena from the imagery supplied. Getting a more specific answer is required. Direct fluid feature extraction has the following advantages over these exploratory visualization tools:

- Deterministic Algorithms -- If there are no "parameters" that the user need adjust, then no intervention is required.

- Fully Automated -- The analysis can be done off-line in a batch computation. It can be used directly by a solver to adapt the mesh to better resolve the feature.

- Local Analysis -- These schemes, where possible, perform only local operations. Therefore, the computations for each cell are independent of any other cell and may be performed in parallel. This is clearly advantageous in distributed memory compute arenas.

- Data Reduction -- The output geometry is several orders of magnitude smaller than the input dataset. This is an important characteristic for the size of a resultant output. High fidelity spatial and temporal results of the feature extraction can be stored on disk for non-interactive co-processing environments. This is usually not possible for the entire transient simulation. A beneficial side effect is that playback is rapid (the extraction process is done and the data has been distilled to salient information). Allowing for the playback within an interactive system gives the user the ability to adjust the viewing angle of the resulting image, this cannot be done with a static movie.

- Quantitative Information -- Precise locations for the flow features are extracted. Also, classification and measures of strength can be reported.

The results of feature extraction can be viewed in a three-dimensional (3D) interactive visualization environment and can be used in conjunction with interactive visualization tools. Feature extraction tools are now being used in parametric studies where tens or hundreds of simulations are run for the same design with subtle changes to the structure or flight conditions. These tools can be used to detect the existence of such features as shocks [1], vortex cores [2], recirculation zones [3], boundary layers [4] and separation and re-attachment lines [5]. All of these feature extraction algorithms have been collected together into a single software toolkit: **FX** [6].

The existence of secondary flow is another feature of significant importance to CFD turbomachinery engineers. The identification of secondary flow can highlight areas of stress and loss. Although the concept of secondary flow is relatively understood, there is no commonly accepted mathematical definition. Because of this it has been extremely difficult to develop an automated feature capability for the identification of secondary flow. This paper will present a formal definition for this concept and one approach for automatically detecting and visualizing secondary flow. In addition to the definition of secondary flow this paper will discuss how such an automated feature extraction utility was developed and used in the post-processing analysis of CFD simulations. Of particular interest is the application of this tool to add insight to the final analysis.

BACKGROUND

The concept of secondary flow in turbomachinery is generally thought of as any flow that is not in the direction of the primary flow. An example of this vague definition is shown in Fig. 1. The vortices shown in the figure are examples of secondary flow as the primary flow direction is directly between the blades. Some attempts at rigorously defining this idea include:

- ...component of absolute vorticity in the direction of the relative streamline [7].
- Secondary flow in broad terms means flow at right angles to intended primary flow [8].
- Due to viscous effects, end walls divert primary flow produced by blades and vanes; to give rise to what has come to be called secondary flow [9].

Of the three definitions listed above only [8] provides a definition that could be made operational. What is required is the notion of primary flow, which we can define. Unfortunately by the time we get a CFD solution the notion of *intended* is lost.

The desire to view perturbations from the *primary* flow direction can give insight into the viscous, reverse flow and vortical effects that deviate from the design. To this end it is obviously desirable to be able to generate two-dimensional vector plots that display the secondary flow given a traditional CFD solution.

Secondary flow plots are usually displayed in a passage between blades or just downstream from the trailing edge. The arrows are generated from a frame of reference that is relative to the passage in question (i.e., absolute for fixed rows and moving for rotors). This points to a difficulty in areas between stators and rotors: what is the appropriate frame of reference? Because of this ambiguity the frame of reference is specified by the user in the final implementation.

ALGORITHM

It would clearly be desirable to have a scheme that could maximize the primary flow through a constructed surface. This could be done by defining a *pivot* point in the channel that reflects the centroid of the passage or flow. A surface that goes through the point can be generated. By adjusting the position of this surface the *best* fit can be found. This surface can then be used to view the secondary flow by projecting the vector field data onto the surface.

The goal here is to calculate the mass-averaged quantities in the channel. This should be done on a grid plane or a cut through the passage that is orientated so that all bounds of the cut are walls (if possible). The following can be done with either a plane (all surface facets have the same normal *n*) or an analytical surface where the normal for cut facets can change.

Compute surface integrals:

$$\overline{A} \equiv \iint \vec{n} dA \quad (1)$$

$$\overline{M} \equiv \iint \left[\rho \vec{q} \cdot \vec{n}\right] dA \quad (2)$$

$$q_0 \equiv \overline{M} / \overline{A} \quad (3)$$

$$\vec{X}_0 \equiv \iint \left[\vec{X} \rho \vec{q} \cdot \vec{n}\right] dA / q_0 \quad (4)$$

where $\overline{A}$ is the area, $\vec{n}$ is the surface normal vector, $\vec{q}$ is the velocity vector and ρ is density. $\overline{M}$ is the mass-averaged flux, q_0 is the mean velocity and $\vec{X}_0$ is the mass-averaged center of the flow.

Newton-like Iteration to Maximize Primary Flow

By selecting various cuts that pass through $\vec{X}_0$ we can adjust the normal (in the case of a simple plane) in an iterative loop so that we maximize q_0 (the velocity perpendicular to the plane):

$$\vec{n} = |q_0|$$

Note that a new set of surface integrals is computed during each iteration. This can also change the position $\vec{X}_0$.

Using a planar cut this technique takes about 3 to 4 iterations to converge (i.e., the normals returned differ by some suitably small factor). This Newton-like convergence is most always seen unless the planar cut is adjusted so that a new portion of the flow field is exposed.

When converged, this provides a view of the data that displays secondary flow when the normal velocity component is removed.

DISCUSSION

In practice this algorithm works well, but did require a number of operational adjustments. These included:

Passage of Interest

The fast cut algorithms are based on Marching Cubes [10] and are performed on a single 3D element at a time. The result is a set of disjoint polygons that reflect the portion of the surface that cuts through the cell. The notion of where in the domain the fragments come from is usually lost. If the simulation contains more than a single passage, the cut data can easily contain fragments from elsewhere in the simulation. This will corrupt the primary flow calculation in that we are no longer focused on a single passage.

The solution is to reconnect the fragments into complete (and bounded) surfaces. Once this is done the location of a seed point can be used to specify which bounded surface to use. Only those polygons that are within the same bounded surface as the specified seed point are used in the calculations.

The cut algorithm used constructs the surface in a Finite-Element sense (that is, a list of nodes that reflect the 3D edges being cut is constructed and the polygons refer to indices in that list). The reconnection is performed via a polygon side-matching algorithm based on the indices (not floating point locations). This is unique and robust. Any side that is seen by two polygons is interior to the region. A side with only a single polygon is bounding the region.

Multi-Block Simulations

In multi-block simulations the volumes represented by the blocks can abut or overlap. The individual cell definitions are usually block specific so that even if the blocks maintain a larger contiguous volume, it is usually not apparent by the time one looks at the fragments from the Marching Cubes results. When reconnecting the regions the results will reflect the block boundaries and not the actual bounds of the cut. The regions need to be placed back together.

When performing streamlining, it is traditional to use the "IBlank" data to inform the software how the blocks are connected. When one pierces a cell on a face where the "IBlank" data indicates that a jump to another block is required, the "IBlank" index contains the accepting block. Initially, this data was used to attempt to flood the region from the target surface fragments to connecting blocks. This was found to be unreliable.

A much more expensive technique was developed. This involved producing a bounding-box around each region as a first step. All regions (that have not been included) and have bounding boxes that overlap the start region are examined. Each point on the exterior of the start region is compared to all fragments of the candidate regions. If it is found that any point is interior, then the new region is considered part of the calculation and this process is then recursively applied where this candidate becomes the start region.

In this way the seed point fills all connecting and overlapping areas and the calculation can proceed on that "passage".

Tip Leakage Simulations

When performing this secondary flow algorithm on a simulation that displays tip leakage there is a natural connection between passages. When using the flood algorithm described above there will be spilling into other connected passages. This will corrupt the primary flow calculation.

This problem has been taken care of if the simulation is multi-block and there are individual blocks that represent the tip flow regions. The flooding can be "dammed" by informing the technique not to use certain blocks as candidates.

Frame of Reference

In multi-stage calculations care needs to be taken so that the algorithm sees data in a consistent frame of reference. This means that when looking at the secondary flow in a rotor, all velocity field values should be in the rotating frame. It is important that the data in the stators be transformed so that the technique does not see any discontinuities in the velocity field.

This then means that if one were to traverse the machine from upstream to down that there will be a number of changes of reference. These should be done while the resultant planar cut is in the zone between blades.

IMPLEMENTATION

The algorithms for the detection of secondary flow features were implemented in C in the **Feature** e**X**traction toolkit **FX** [6]. This toolkit, unlike most visualization systems, is lightweight because no drawing and/or GUI functions are supported. In general, the input is the CFD solution and output is various forms of geometry. The following entry point has been added:

```
FX_MEANFLOW(XPOS, VNORM, DAM)
```

`float XPOS[3]` -- On input this is the position that sets the plane given the normal `VNORM`. On output, the mass averaged position on the planar cut is returned.
`float VNORM[3]` -- On input this is the normal that sets the family of planes to use to produce the cut. On output `VNORM` is filled with the mass averaged velocity through the cut.
`int *DAM` -- Pointer to the status of each block (for multi-block cases) to act as a "dam" for the flooding procedure. Zero indicates that flooding through the block is acceptable; a one is the flag to not use this block. NOTE: may be `NULL` to specify no "damming" or for non multi-block cases.

This subroutine, given the start position and plane normal, computes the mass-averaged "center" of flow and the mass averaged velocity. This is essentially a single iteration in the maximization of the primary flow through the surface. Fixing the pivot point is controlled by the position specified in `XPOS`. The position can be made stationary by resetting the values to the fixed position after each iteration. This will allow for the examination of secondary flow in many contexts (for example, picking points along a streamline, one can get the sense of the secondary flow as seen by a traveling fluid "particle").

A GUI was developed using C++, OpenGL, and QT to interact with **FX** for the specification and eventual visualization of the secondary flow features. This system was used to generate all of the images shown in this paper.

The ability to calculate and display secondary flow vectors was added to an interactive visualization system. The initial input to the system are the names of the geometry and solutions files and a specification of the surfaces that are to be displayed. In this manor the reference geometry of a simulation is generated and displayed before any additional visualization is done. One such initial image is shown in Fig. 2. The user has the capability to adjust the view of the image by translating, rotating, and scaling the image. In order to calculate the secondary flow features an additional panel is used. This panel is shown in Fig. 3. The user can specify the location of the pivot point and the normal direction of the initial plane. Figure 4 shows an initial specification of the secondary flow plane. The plane is displayed in blue and the yellow cross hair indicates the initial pivot point. From this panel the user then initiates the calculation of the primary flow plane. Once the resulting image appears the user can then adjust how the secondary flow vectors are displayed. The size of the display grid, the number of grid points and the vector length factor can all be modified through this panel. What images are displayed in the viewing window are also specified with this panel. The display of the initial plane and the pivot point can be turned on and off. The user also has control of the display of the contour plane and the flow vectors. The color of the vectors and the tuft points can also be modified. Figure 5 shows the calculated secondary flow plane in its position relative to the two blades. Figure 6 shows a close up view of the calculated secondary flow plane. The contours shown on the plane are those of density.

It is also possible to generate an MPEG animation by using this system. Multiple frames are generated by specifying a path (either using physical coordinates or computational coordinates) where the initial pivot points are to be placed for each image. The panel used to specify these parameters is shown in Fig. 7. In addition to the parameters specified in the panel shown in Fig. 3 the user is given the option to have the resulting flow plane placed in the center of the screen parallel to the viewer.

RESULTS

This system was first tested on the results of an analysis for an injector chamber, and then used to analyze the results of a simulation of an axial turbine from a turbopump. Figure 8 shows the initial geometry of the injector chamber and the injector holes. This simulation was done on one quarter of the complete chamber. Figures 9 and 10 show the secondary flow vectors projected onto a density contour surface near the injectors. An animation was generated that showed how the secondary flow diminished as the flow moved down the chamber. The application of this system to the injector chamber is important because the amount of secondary flow and swirl (i.e., flow mixing) within the chamber controls the burning characteristics within the chamber.

The system was also used on a distinctly different geometry that of an axial-flow turbine stage from a notional rocket turbopump. Figure 11 shows three images that highlight the flow through one rotor passage of the turbine. The image on the left is a vector plot that shows the direction of the primary, or core flow. The image in the middle is the result of calculating the plane that accounts for the primary flow at a location near midchord of the rotor. This plane contains the projected vectors (overlaid on contours of density) that represent the secondary flow in this region. The vectors are anchored at the black tuft locations and the direction of flow is away from the tuft. The image on the right is a close up view of the secondary flow. The flow in this image is consistent with the expected results found in the diagram shown in Fig. 1. Both figures show a horseshoe vortex, which emanates from the hub endwall. In addition, Fig. 11 highlights the tip clearance leakage vortex. Figure 12 shows a similar set of results further downstream. The position of the plane is nearly perpendicular to the surface of the rotor. The secondary flow vectors shown in the right image clearly show the tip clearance leakage vortex and movement of the endwall vortices towards midspan. An animation was also created that shows the secondary flow for a series of pivot locations along the suction side of the rotor blade. Figure 13 shows one frame from this animation. The image on the left shows both a stator and a rotor blade to establish the location of the primary flow plane within the simulation. The image on the right is a close up view of the resulting primary flow plane displaying secondary flow vectors.

CONCLUSIONS

A set of tools has been developed to aid in the detection and visualization of secondary flow features in CFD results. The interactive tool provides for a platform to display several aspects of the geometry and the flow solution. The ability to determine the primary flow direction and then display secondary flow features has been found to be helpful in the analysis of an injector chamber and an axial flow turbine.

AKNOWLEDGEMENTS

Portions of his work was performed under NASA grant NAG8-1872.

REFERENCES

1. Lovely, D. and Haimes, R., “Shock Detection from Computational Fluid Dynamics Results”, AIAA Paper 99-3285, June 1999.

2. Sujudi, D. and Haimes, R., “Identification of Swirling Flow in 3-D Vector Fields”, AIAA Paper 95-1715, June 1995.

3. Haimes, R., “Using Residence Time for the Extraction of Recirculation Regions”, AIAA Paper 99-3291, June 1999.

4. Basket, L, and Haimes, R., “Feature Extraction of Shear Layers”, AIAA Paper 2001-2665, June 2001.

5. Kenwright, D., “Automatic Detection of Open and Closed Separation and Attachment Lines”, Proceedings of IEEE Visualization ‘98, October 1998.

6. Haimes, R., “**FX** – **F**luid feature e**X**traction tool-kit”, http://raphael.mit.edu/fx.

7. Hawthorne, W. R., Cambridge University CUED/A-Turbo/TR 63, 1974.

8. Cumpsty, N. A., ‘Compressor aerodynamics’ Longman ISBN 0-582-01364-X page 316, 1989.

9. Bradshaw, P., 1996, The Bradshaw quote is from the paper by L.S. Langston (reference 2) ‘Secondary Flows in Axial Turbines - A Review’ annals New York Academy of Sciences, May 2001.

10. Lorensen, W., and Cline, H., “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”, Computer Graphics (SIGGRAPH 87 Proceedings) 21(4) July 1987, p. 163-170.

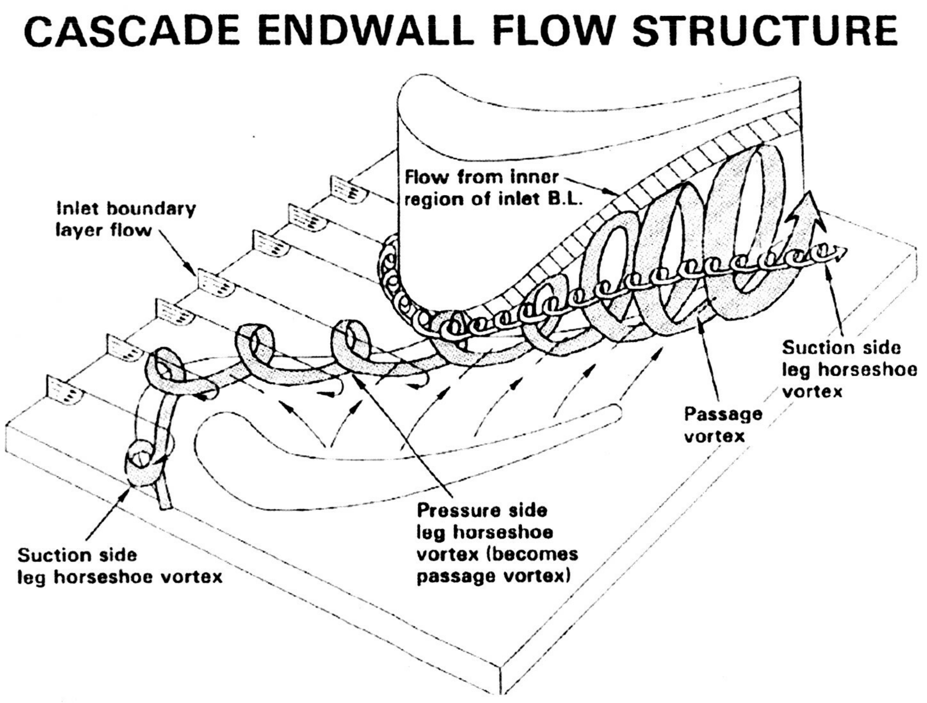

CASCADE ENDWALL FLOW STRUCTURE
Flow from inner region of inlet B.L.
Inlet boundary layer flow
Suction side leg horseshoe vortex
Passage vortex
Pressure side leg horseshoe vortex (becomes passage vortex)
Suction side leg horseshoe vortex

Figure 1: Example of Secondary Flow

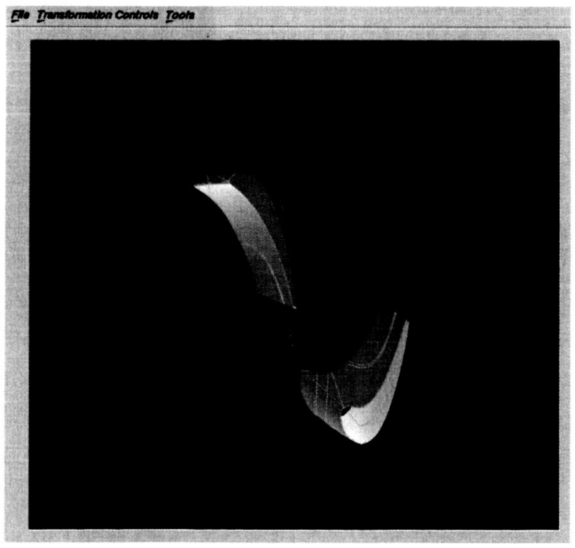
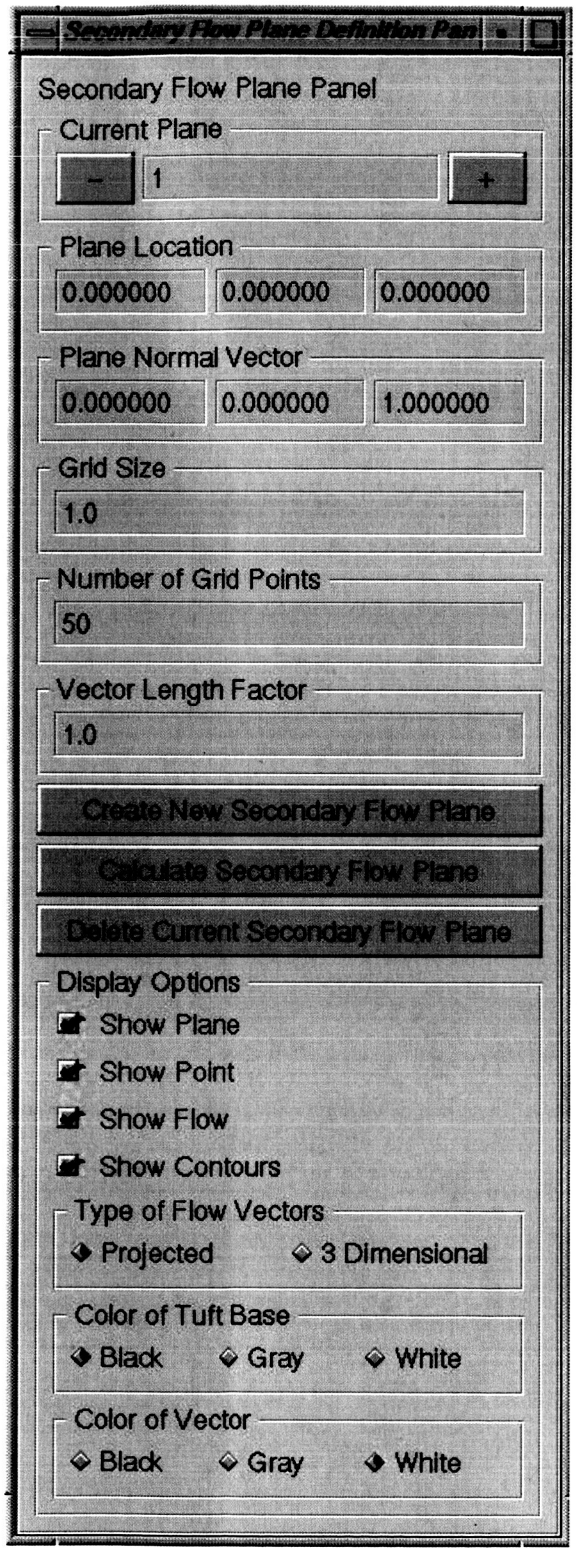


Figure 2: Initial Image



Secondary Flow Plane Definition Pan

Secondary Flow Plane Panel

Current Plane: 1

Plane Location: 0.000000 0.000000 0.000000

Plane Normal Vector: 0.000000 0.000000 1.000000

Grid Size: 1.0

Number of Grid Points: 50

Vector Length Factor: 1.0

−
+
Create New Secondary Flow Plane
Calculate Secondary Flow Plane
Delete Current Secondary Flow Plane

Display Options

- [✓] Show Plane
- [✓] Show Point
- [✓] Show Flow
- [✓] Show Contours

Type of Flow Vectors: [✓ Projected] 3 Dimensional

Color of Tuft Base: [✓ Black] Gray White

Color of Vector: Black Gray [✓ White]

Figure 3: Secondary Flow Control Panel

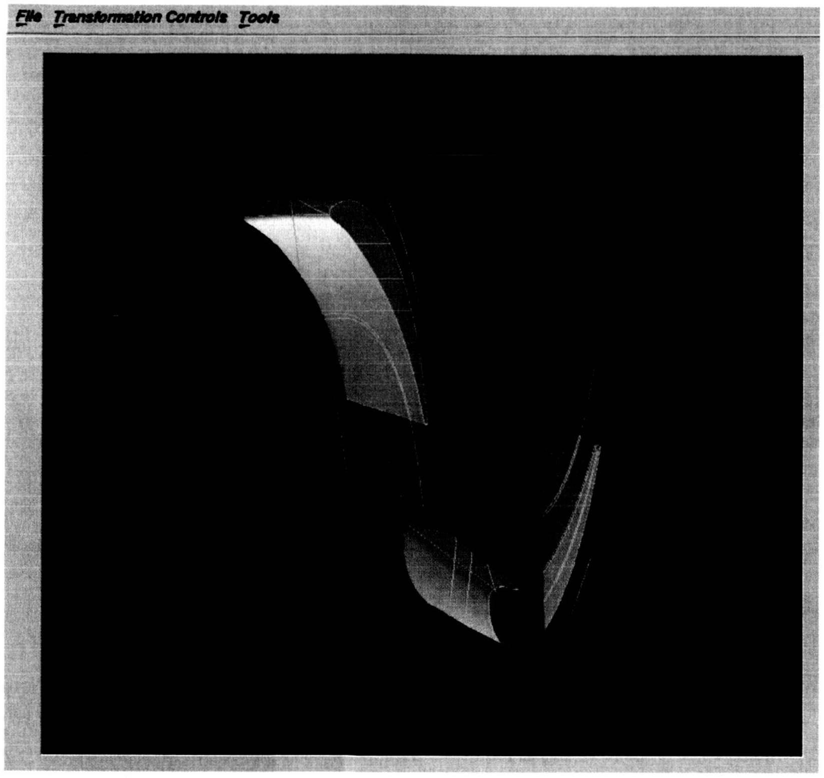


Figure 4: Initial Position and Pivot Point of Secondary Flow Plane

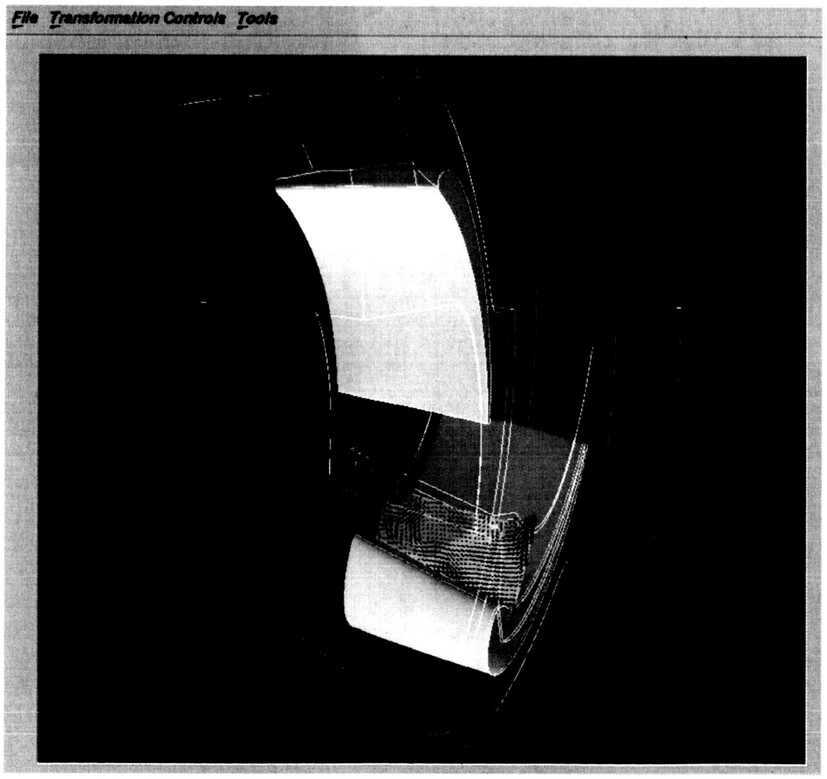


Figure 5: Calculated Secondary Flow Plane

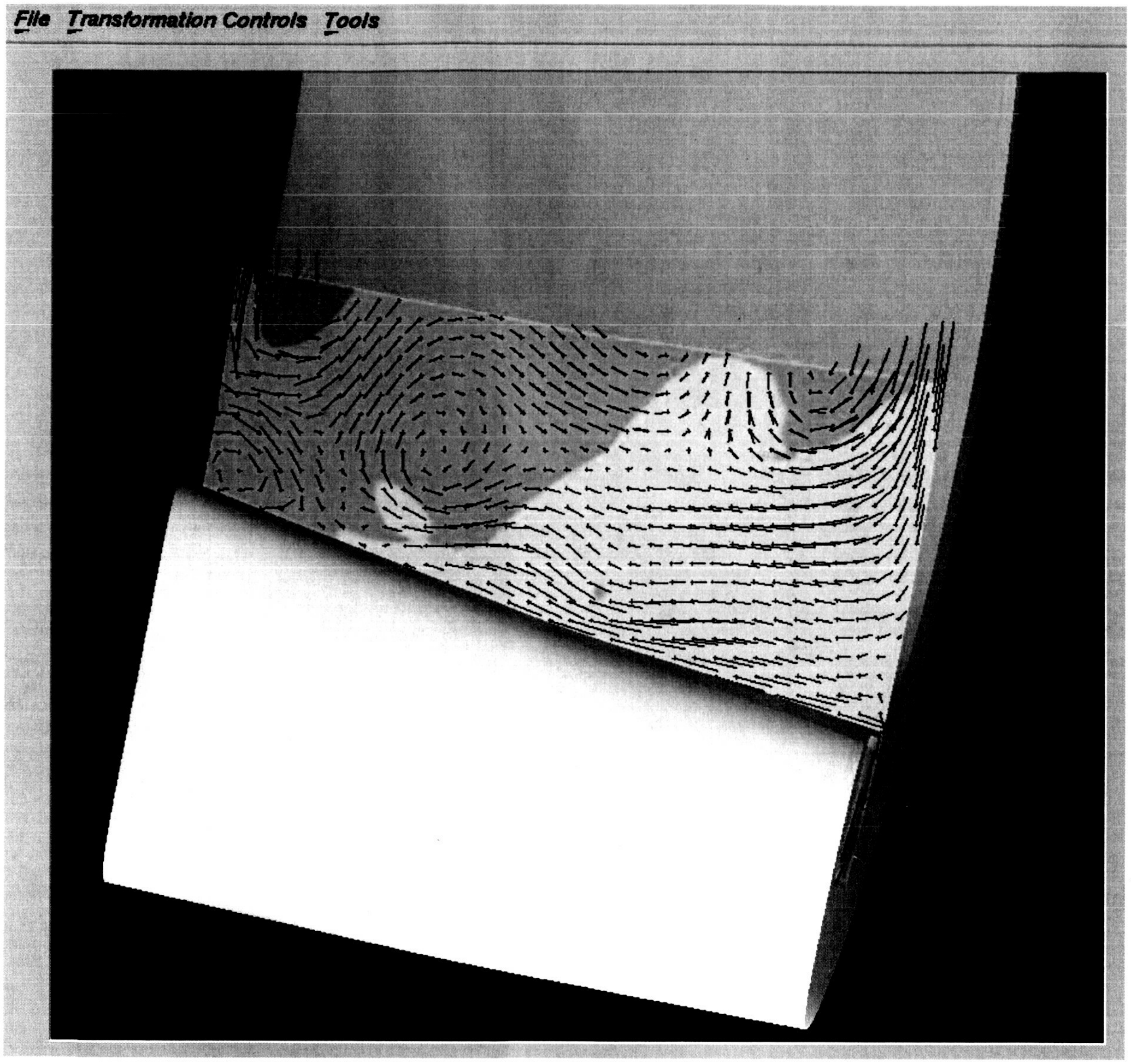
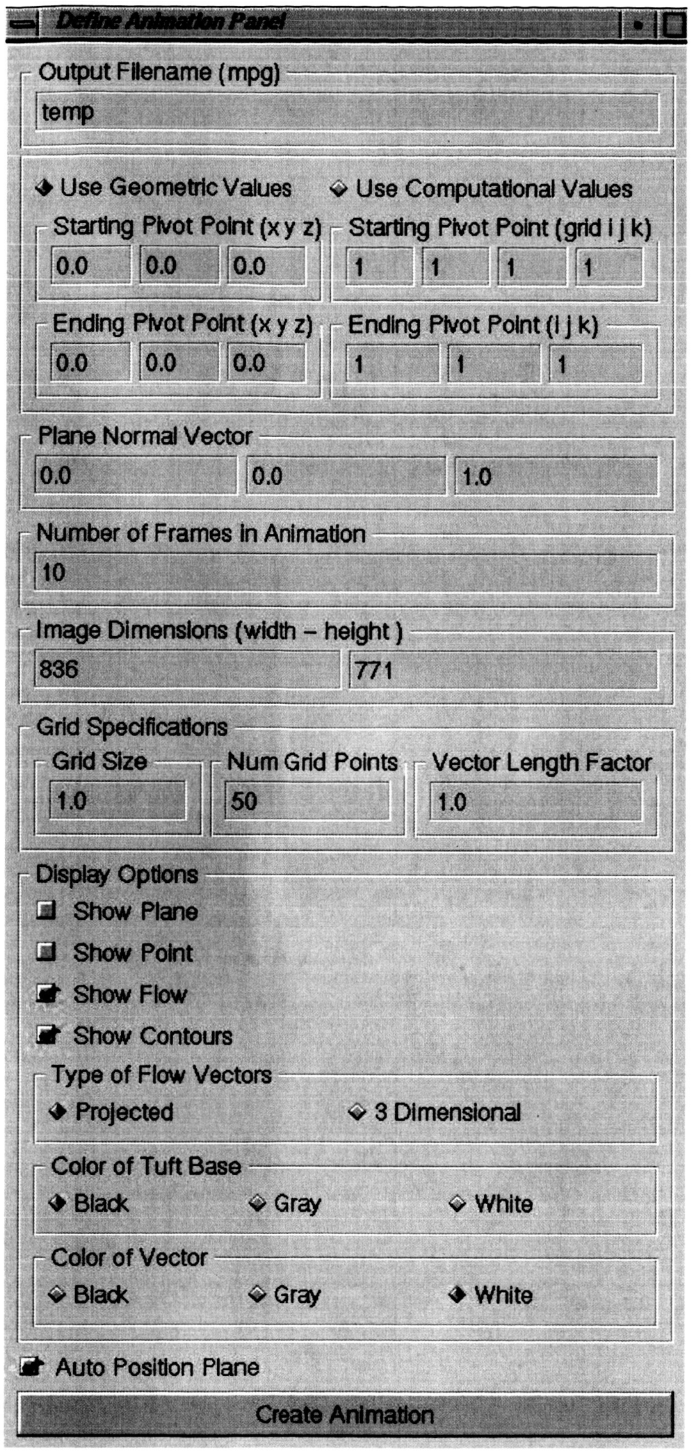


Figure 6: Close Up View of Vortices


Define Animation Panel

Output Filename (mpg): temp

[✓ Use Geometric Values] Use Computational Values

Starting Pivot Point (x y z): 0.0 0.0 0.0
Starting Pivot Point (grid i j k): 1 1 1 1

Ending Pivot Point (x y z): 0.0 0.0 0.0
Ending Pivot Point (i j k): 1 1 1

Plane Normal Vector: 0.0 0.0 1.0

Number of Frames in Animation: 10

Image Dimensions (width – height): 836 771

Grid Specifications
- Grid Size: 1.0
- Num Grid Points: 50
- Vector Length Factor: 1.0

Display Options
- ☐ Show Plane
- ☐ Show Point
- ✓ Show Flow
- ✓ Show Contours

Type of Flow Vectors: [✓ Projected] 3 Dimensional

Color of Tuft Base: [✓ Black] Gray White

Color of Vector: Black Gray [✓ White]

✓ Auto Position Plane

Figure 7: Animation Control Panel

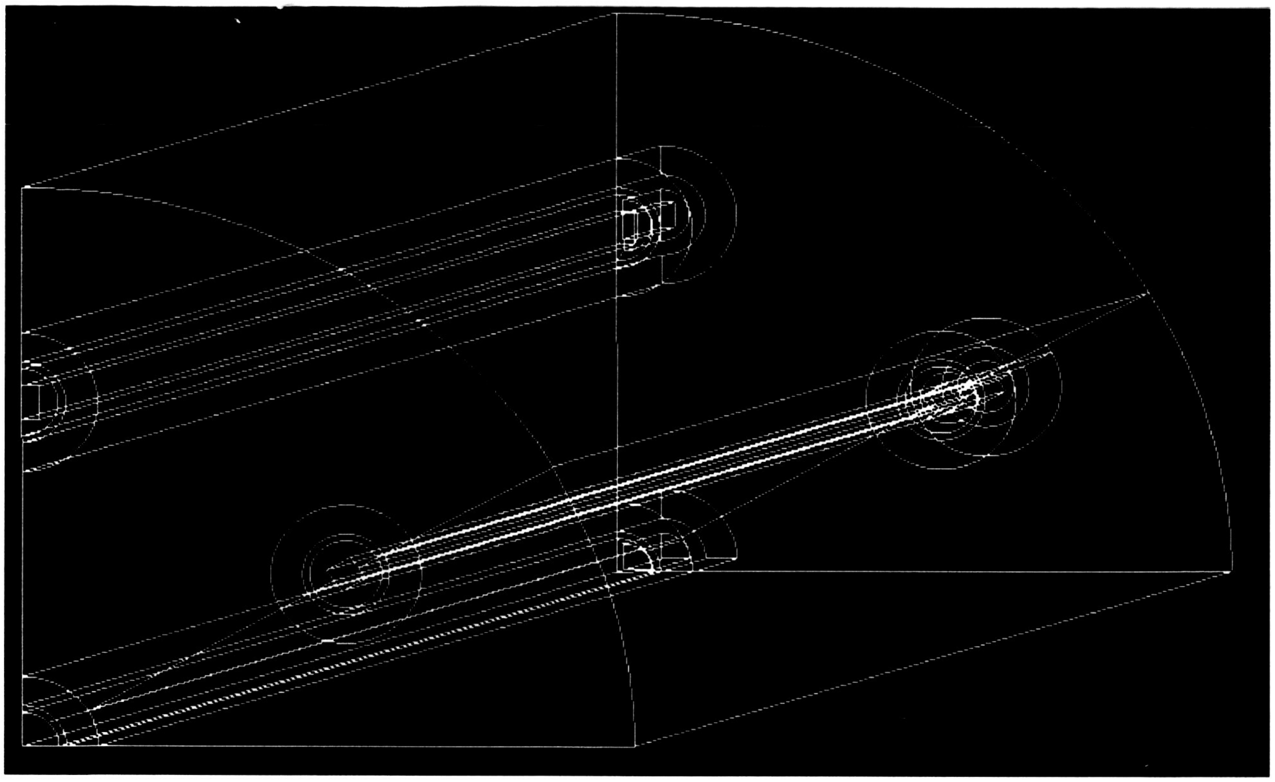


Figure 8: Geometry of Injector Chamber

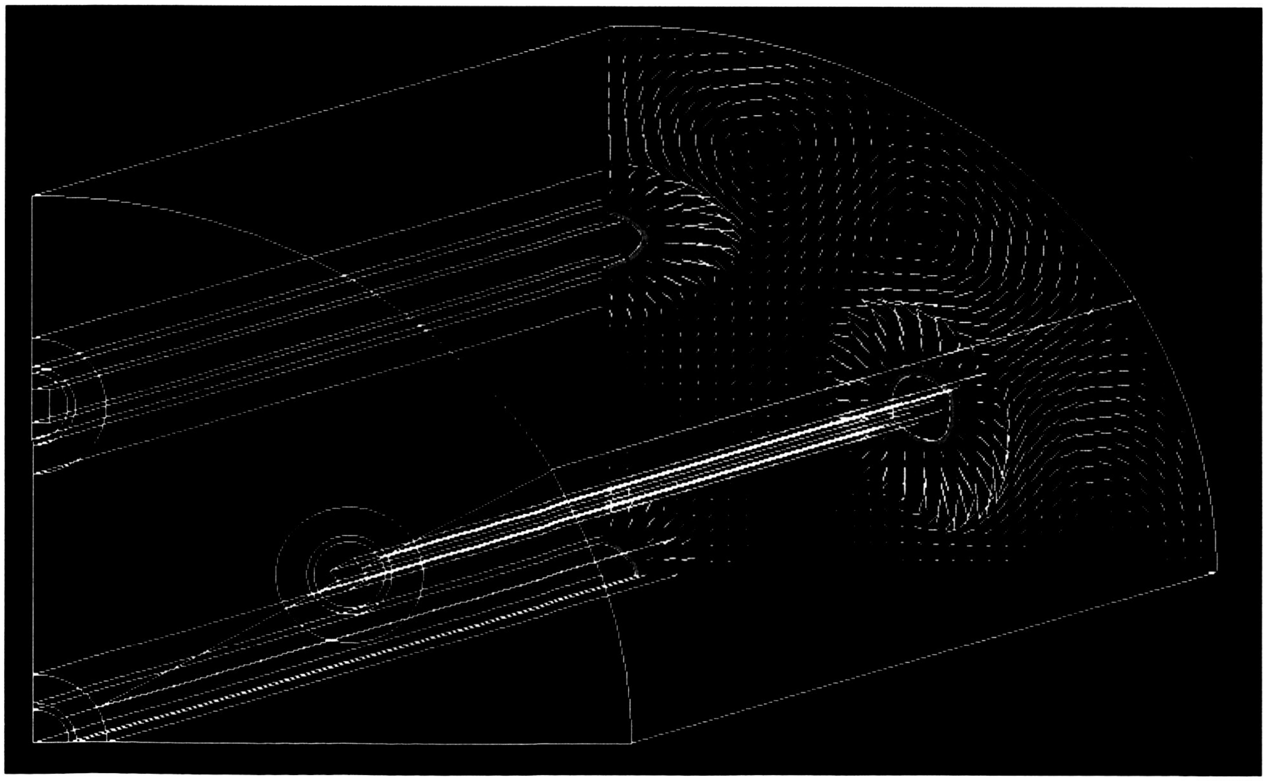


Figure 9: Secondary Flow Vectors Projected onto Density Contour Surface

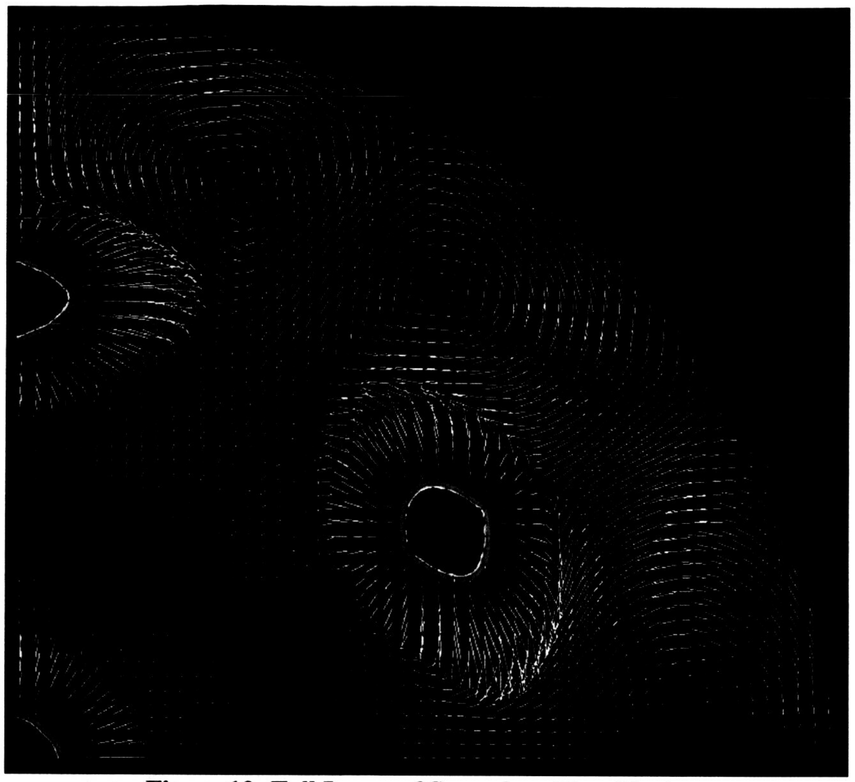


Figure 10: Full Image of Secondary Flow Vectors

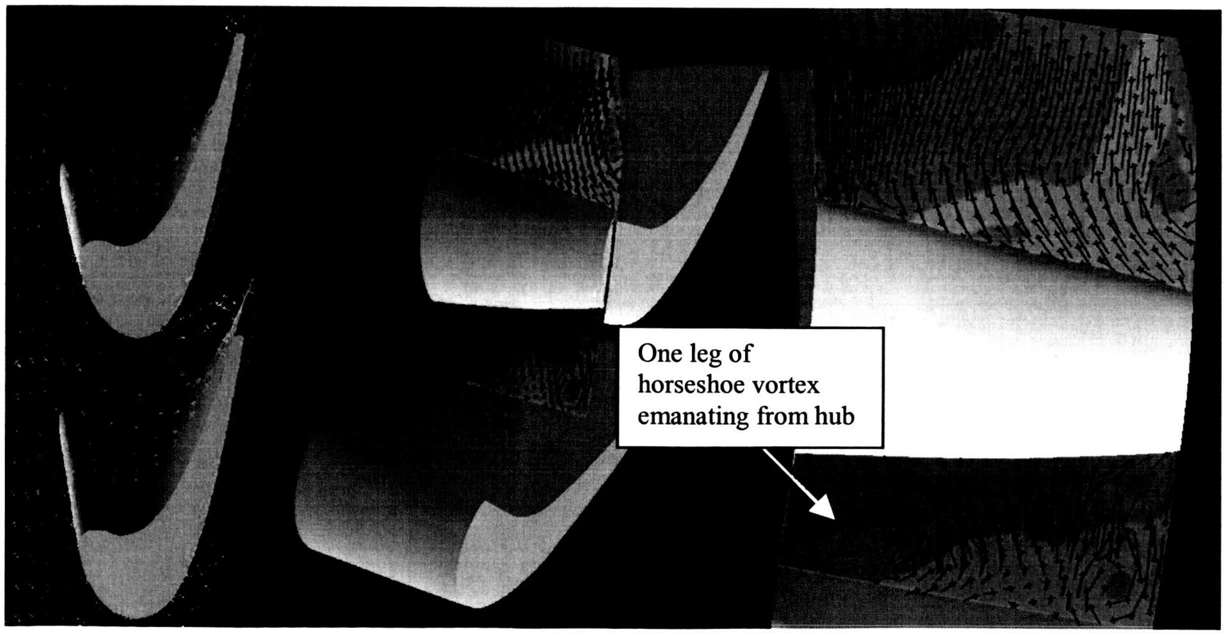


Figure 11: Vectors Showing the Primary flow Direction (left), Position of Secondary Flow Plane (middle) Secondary Flow Image (right)

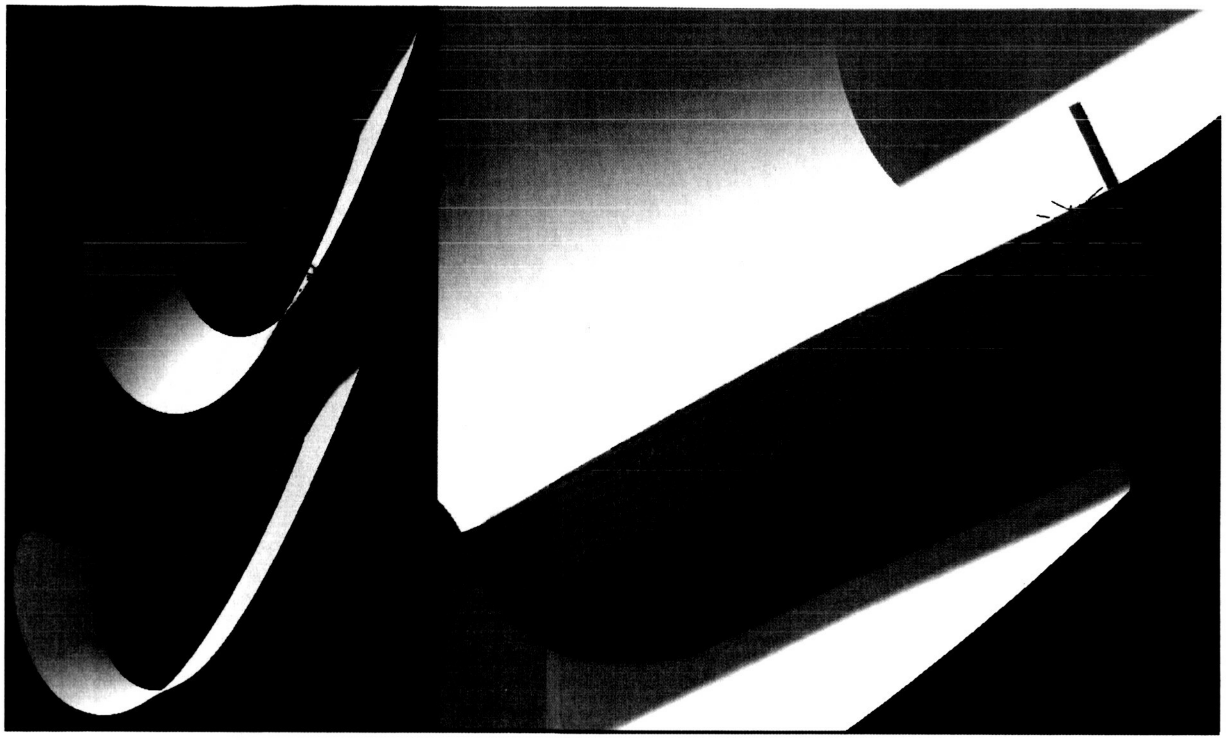


Figure 12: Position of Primary Flow Plane (left) and Close Up View of Secondary Flow Results (right)

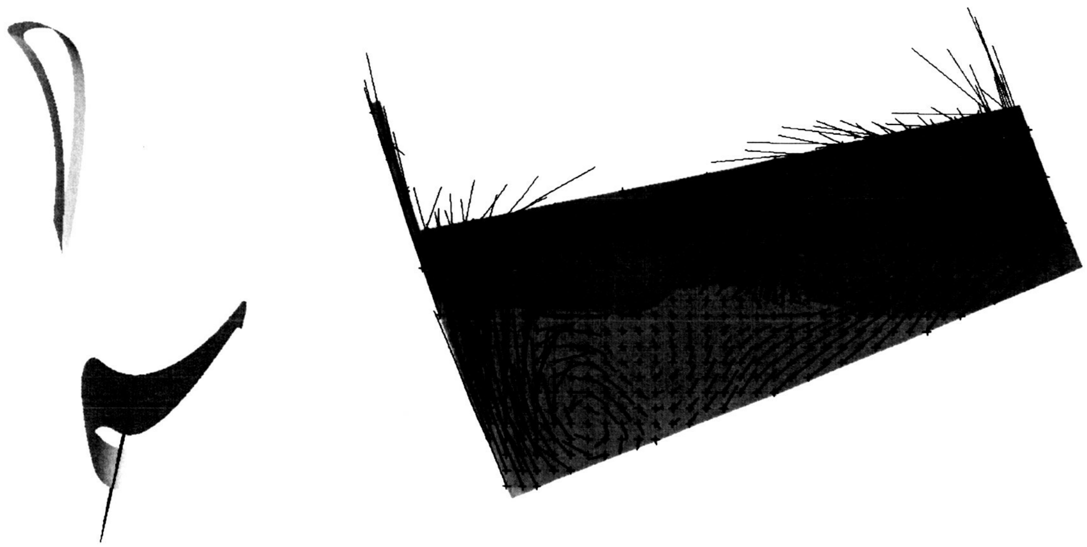


Figure 13: Image from Secondary Flow Animation

Rendering Planar Cuts Through Quadratic and Cubic Finite Elements

Michael Brasher* Robert Haimes†

Aerospace Computational Design Laboratory
Massachusetts Institute of Technology

ABSTRACT

Coloring higher order scientific data is problematic using standard linear methods as found in OpenGL. The visual results are inaccurate when there is a large scalar gradient over an element or when the scalar field is nonlinear. In addition to shading nonlinear data, fast and accurate rendering of planar cuts through parametric elements can be implemented using programmable shaders on current graphics hardware. The intersection of a planar cut with geometrically curved volume elements can be rendered using a combination of selective refinement and programmable shaders. This hybrid algorithm also handles curved 2D planar triangles.

CR Categories: G.1.8 [Numerical Analysis]: Partial Differential Equations—Finite Element Methods; I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, Shading, Shadowing, and Texture;

Keywords: Higher Order Elements, Programmable Shaders, Cutplanes

1 INTRODUCTION

Numerical methods are widely used throughout academia and industry to solve physical problems when experimental data is difficult to obtain. The details of these methods can vary greatly, but they all essentially solve a set of governing equations by discretizing the domain of interest and solving an analogous formulation at the discrete points or nodes. Once a solution has been generated for these nodes, then data over the entire domain can be obtained by interpolation. The simplest way to interpolate is to assume linearity within each cell based on the vertices that support that element. There are a number of ways available to then view this data, since most visualization techniques are based on the assumption of linear interpolation. However, there are many situations in which it is advantageous to solve the discrete equations using a non-linear basis or higher order elements [4, 10]. This can mean using anything from the polynomial Lagrange basis to a hierarchical basis or spectral elements. One obvious difficulty with using higher order numerical methods is that there is no simple way to visualize the data in its native form (since most current visualization software uses a linear basis). This renders higher order methods much less useful. Understanding of numerical results and new insight is often only possible when one can accurately visualize the massive amounts of data produced.

Accurate rendering of nonlinear data cannot be performed efficiently using only the standard OpenGL API, since all OpenGL primitives are inherently linear. Higher order data can be interpolated and rendered quite simply and quickly by utilizing the flexibility of modern graphical processing units (GPUs). In addition to rendering surfaces, one important technique used in scientific visualization is the generation planar cuts through 3D field data. This can be accomplished through a combination of selective refinement of the elements and accessing programmable shaders inside the GPU.

*mbrasher@mit.edu
†haimes@mit.edu

2 PREVIOUS WORK

3D graphics APIs like OpenGL are designed to use planar primitives because of the simplicity of the resulting algorithms. This ability to render linear elements can be leveraged to visualize nonlinear surfaces through polygonization, which essentially translates the higher order surface into one that is piecewise linear. This method was used in [1] to render parametric surfaces, while an adaptive refinement method was used in [11] to subdivide implicit surfaces. This was then generalized to handle both implicit and parametric surfaces with a multi-resolution hierarchical structure in [12]. These methods are able to sample the higher order data in way that can be handled by traditional visualization algorithms (i.e. at the end linear elements are produced).

A hierarchical approach was also used by [6] and [13] in the direct visualization of higher order data. In [13], volume visualization was accomplished by ray casting through both straight-edged and curved quadratic elements. Isosurface extraction was performed by approximating the surface by quadratic patches in parameter space, transforming them to physical space, and rendering the resulting quartic functions through higher order patch rendering in hardware. Texture shaders and register combiners were used in [7] to visualize higher order hexahedra. The hardware limitations of using texture shaders and register combiners can be avoided by instead using a fully programmable shading language like *Cg* [3].

3 DISCONTINUOUS FEM

One popular group of numerical techniques, the Finite Element Methods (FEM), are particularly convenient when dealing with complex geometries or unstructured computational meshes [10]. FEM simplifies the solution scheme by mapping every element in the mesh to a master *reference* element, and then scalar interpolation can be performed using shape functions as a basis. Regardless of the basis used in the computational solver, the data can be easily converted to any other basis of the same order, so only the Lagrange basis will be discussed. Furthermore, only simplicial elements will be considered.

When rendering continuous data, neighboring elements share both the location and field data of common nodes. The use of collected primitives (polytriangles, quad meshes and etc.) can speed up the display time since the support data needs to be passed along the graphics pipeline fewer times. However, the direct goal of this research was to visualize flow solutions generated using the Discontinuous Galerkin (DG) method [4], [2]. As such, any scheme developed should be able to naturally handle discontinuities (at element faces) in the scalar fields being visualized. The simplest way to accomplish this is for each element to independently store data for all of its basis nodes, similar to [8]. Even though the physical location of shared nodes is the same between neighboring elements, nodes must be respecified for each element in which they appear. The goal is to have a method that allows for easy handling of both continuous and discontinuous data with the acknowledgement that there will be some lose of the speed benefits in comparison to the use of collected primitives for continuous data.

3.1 Reference Element Interpolation

In general, a triangular element T has a scalar interpolant of order p and q degrees of geometrical freedom. The degrees of freedom determine if and how the sides of T are curved, and the order of interpolation determines how many nodal values of the scalar function are needed to specify the interpolant. For example, a p_3q_2 triangle would have a cubic polynomial scalar interpolant and quadratic geometry.

Using the Lagrange basis, every element in the mesh can be mapped to a reference element. The reference coordinates, $\bar{\xi}$, are aligned so that the component ξ_i is 1 at vertex i of the reference element and 0 at all other vertices. Note that there are 3 reference coordinates in 2D and 4 reference coordinates in 3D. The extra degree of freedom is removed by requiring that the coordinates identically sum to 1, i.e. $\sum_i \xi_i = 1$. The nodal shape functions ϕ_i are defined so that at each node n_j:

$$\phi_i(n_j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{1}$$

Given a scalar function with nodal values s_i at node n_i, the value of the scalar interpolant $s(\bar{\xi})$ at a point $\bar{\xi}$ is given by:

$$s(\bar{\xi}) = \sum_i s_i \phi_i(\bar{\xi}) \tag{2}$$

It is convenient to scale the nodal values so that the scalar interpolant is contained in $s \in [0,1]$. Once the value of the scalar interpolant is found at a point, the color at that point is defined by some arbitrary colormap. One standard choice of a colormap is the spectral colormap shown in fig. 1.

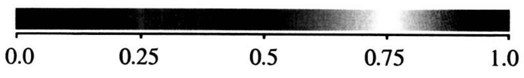


Figure 1: Spectral Colormap

In addition to nonlinear scalar data, the geometry of the element can be curved. Only the coordinates of each node in physical space, $p_i = \{x_i, y_i, z_i\}$, need to be specified, and then the geometry of the element is interpolated in the same manner as the scalar field using eq. 2. As a matter of practice in computational meshes, there will be $q > 1$ elements conforming to the curved boundaries and linear $q = 1$ elements on straight boundaries and in the interior. At times $q > 1$ interior elements may be seen when there is a stretched mesh near a curved boundary. This ensures positive volumes and well-behaved interpolation.

3.2 Dimensional Hierarchy

Given physical coordinates at the nodal points, the p_x reference elements map to some curved region in physical space, called a p_x tetrahedron in 3D, a p_x triangle in 2D, and a p_x line in 1D. The four faces of a p_x tetrahedron can be mapped to the 2D reference element, so each face can be described as a p_x triangle. Similarly, the three edges of a p_x triangle can be described as a p_x line. Thus the simplicial elements form a dimensional hierarchy where a p_x simplex of dimension n contains p_x simplices of dimension $n-1$.

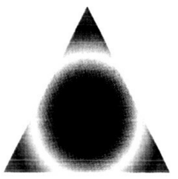


Figure 2: p_2 Shader

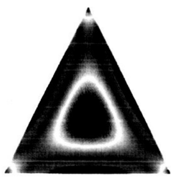


Figure 3: p_3 Shader

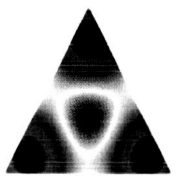


Figure 4: p_4 Shader

This concept of a dimensional hierarchy is not restricted to the faces and edges. Any planar polygon in the 3D reference space can be triangulated into curved triangles, and any line segment in the 2D reference space can be described as a higher order line. However, not all curved regions can be described as a p_x line, triangle, or tetrahedron. Any nonlinearity in the reference space will be compounded in the mapping, and the resulting interpolation will not be p_x.

4 Shading Parametric Elements

In order to visualize a parametric element with scalar values, s_i, at each node, eq. 2 must be implemented in some manner. OpenGL alone can only do this by refining the triangle or generating a texture map. Both of these methods become extremely slow as the number of triangles increases. An alternative is to use the programmability in the GPU exposed by graphics languages like *Cg*. This is where great performance gains can be obtained. The GPU can inherently use the parallelism in these operations because the rasterization phase generates a pixel at a time (with no dependence on neighboring pixels). The processor can parcel out each pixel in the fragment to the number of raster engines available in the specific graphics hardware.

Eq. 2 can be implemented in a fragment shader by defining texture coordinates at each vertex as the vertex's position in reference space, $\bar{\xi}$, and then evaluating the shape functions in the fragment shader. The results of this shader on one triangle is shown in fig. 2. Figs. 2, 3, and 4 show the results for the p_2, p_3, and p_4 shaders respectively. Note that Gouraud coloring would produce a constant color triangle for each case.

Because Gouraud shading interpolates in color space, coloring artifacts are seen when using the traditional OpenGL pipeline to render triangles with large gradients. This problem is avoided in the fragment shader because full scalar interpolation (even for p_1) is performed and the color applied as a last step via the colormap data.

4.1 Performance

Evaluating the p_2 interpolation in the fragment shader involves more work than standard Gouraud shading. But as the number of vertices in the scene increases, the cost of transforming the vertices (which in most cases cannot run in parallel) overwhelms the extra cost of the fragment shader. As shown in fig. 5, when drawing 4050 triangles, Gouraud shading is 4 times faster than p_2 interpolation done in a programmable shader, but when the number of triangles is increased to 129600, Gouraud shading is only slightly faster than *Cg*. When drawing 4050 triangles, 1 level of refinement is faster than the *Cg*. When drawing more triangles however, the programmable shader is faster than 1 level of refinement and orders of magnitude faster than higher levels of refinement. Considering that programmable shaders are as accurate as refining to the pixel level, it is clear that programmable shaders represent a significant improvement in visual accuracy while running at nearly the same speed as standard linear shading. This is the compelling argument

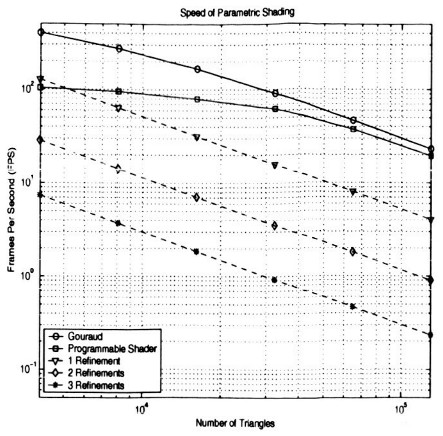


Figure 5: Performance of p_2 Interpolation

for the customized use of GPUs in handing non-linear interpolation. Note: the run times were generated on a P4 2.53 GHz processor with 1Gb of memory and an nVidia GeForceFX 5800 ultra graphics card running under LINUX.

5 Cutplane Intersection

Consider the analytical description of the intersection of a plane cutting through a geometrically curved q_2 or q_3 parametric domain. This intersection is the union of intersections with each individual element, so the problem can be simplified to finding the cutplane intersection with a single element. The discussion will focus on q_2 and q_3 elements, but the method extends naturally to higher orders of parametric elements.

A convenient way to describe the cutplane is by some point p_0 on the plane and the normal to the plane n. Then, the signed distance d of any point p to the plane is given by

$$d = (p - p_0) \cdot n \tag{3}$$

The distance d_i from the nodal points to the cutplane is calculated, and this distance can then be interpolated at any point. Thus, the intersection of the surface and the plane is the locus of points $\bar{\xi}$ that satisfy the equation $d(\bar{\xi}) = 0$.

5.1 Selective Refinement

In order to accurately visualize nonlinear data, the interpolation must be sampled at some set of discrete points. Since the intersection can be described implicitly, it could be polygonized using the method of [11]. While this technique accurately samples a general implicit surface, it does not take advantage of the fact that the intersection is planar.

The simplest approach is uniform refinement (UR), which homogeneously subdivides the q_2 element, and then treats subelements as linear by passing them to the standard marching cubes algorithm [9]. However, as suggested by [5], this can be improved upon given an element T with nodal values s_i, since the scalar field can be bounded. Start by defining:

$$\begin{array}{ll} s_{min} = \min_i s_i & s^- = \frac{1}{2}(s_{max} - s_{min}) \\ s_{max} = \max_i s_i & s^+ = \frac{1}{2}(s_{max} + s_{min}) \end{array} \tag{4}$$

then

$$\begin{aligned} s(\bar{\xi}) - s^+ &= \sum_i (s_i - s^+)\,\phi_i(\bar{\xi}) \\ &\leq \sum_i (s_{max} - s^+)\,\phi_i(\bar{\xi}) \\ &= s^- \sum_i \phi_i(\bar{\xi}) \\ &\leq s^- \max_{\bar{\xi}\in T} \sum_i \phi_i(\bar{\xi}) \end{aligned}$$

taking absolute values

$$|s(\bar{\xi}) - s^+| \leq \left| s^- \max_{\bar{\xi}\in T} \sum_i \phi_i(\bar{\xi}) \right| \leq s^- \max_{\bar{\xi}\in T} \sum_i |\phi_i(\bar{\xi})| \tag{5}$$

and noting that for the 3D q_2 shape functions,

$$\max_{\bar{\xi}\in T} \sum_i |\phi_i(\bar{\xi})| = 2 \tag{6}$$

which leads to the bounds

$$s_{min} - s^- \leq s(\bar{\xi}) \leq s_{max} + s^- \quad ,\forall \bar{\xi} \in T \tag{7}$$

For the 3D q_3 shape functions,

$$\max_{\bar{\xi}\in T} \sum_i |\phi_i(\bar{\xi})| = \frac{16 + 5\sqrt{5}}{9} \leq 3.021 \tag{8}$$

which leads to the bounds

$$s_{min} - 2.021 s^- \leq s(\bar{\xi}) \leq s_{max} + 2.021 s^- \quad ,\forall \bar{\xi} \in T \tag{9}$$

The cutplane M will intersect T if $d(\bar{\xi}) = 0$ at some point $\bar{\xi}$ inside the element. If 0 lies outside the bounds, then T is not intersected, but since the bounds of eqs. 7 and 9 are not tight, T is not necessarily intersected just because 0 is inside the bounds. Still, whether or not $d = 0$ lies outside the bounds can be used as an effective criterion to reject or further refine in a linear selective refinement (LSR) scheme, which treats the final subelements as linear just as in UR. LSR is a more efficient algorithm, since it refines coarsely away from the intersection, and thus handles many fewer subelements. By themselves, eqs. 7 and 9 only dictate whether the element should be refined, they do not specify how. The simplest method is to break the element into equal pieces, and then reapply the bounds to the subelements. However, a more sophisticated adaptive refinement algorithm that seeks to refine where the gradients in the scalar field are highest[6] could be applied.

Even this algorithm is problematic, since the rendering time of LSR is $O(V)$ where V is the total number of vertices that are sent through the graphics pipeline. In order to achieve visual accuracy the refinement must be taken to essentially the pixel level as can be inferred from the simpler results seen in fig. 5. An alternative is to utilize the parallel nature of current graphics hardware by performing the necessary data sampling in the programmable shaders. This allows the nonlinear data to be resolved to the pixel level while sending much less data down the pipeline.

6 Shadow Method for Cutplane Rendering

Though there is a great deal of flexibility when dealing with individual fragments through it Cg, OpenGL is still constrained in the construction of geometry. All pixels passed to the fragment program are a result of the rasterization of a planar primitive. Let such a linear primitive which lies in M and which will completely cover the intersection I be the *shadow* of the intersection. The two main questions to answer are how to generate a shadow and how to shade the fragments in the shadow.

6.1 Determining The Shadow

The shadow primitive should completely cover the intersection so that there are no gaps seen in the final image. Finding a reasonably small shadow is more important than finding the absolute minimal area. The result of too large a shadow is that many pixels will be discarded, which entails additional work in the GPU. The additional effort of finding a smaller shadow must be balanced with the benefit of sending fewer fragments through the GPU's pipeline.

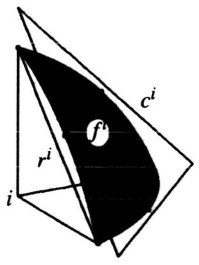


Figure 6: q_2 Orthographic View

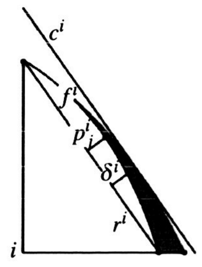


Figure 7: q_2 Side View

To generate the shadow, the curved element is first bounded with a *congruent* q_1 tetrahedron C. Call the linear tetrahedron defined by the four main vertices of the element the *reduced* tetrahedron R. Each face of C will be parallel to R, as shown in fig. 6 and fig. 7. For each main node i of the element, the opposite face is f^i, the corresponding face of the reduced tetrahedron is r^i, and the parallel face of the congruent tetrahedron is c^i. C can then be described by the distance δ^i that each c^i is offset from each r^i, as shown in fig. 7. If $\delta^i \geq 0$, then C always completely contains R. Finding the minimal values of δ^i would require solving:

$$\delta^i = \max_{\bar{\xi}\in f^i} p^i(\bar{\xi}) \quad \text{such that} \quad d(\bar{\xi}) = 0 \tag{10}$$

where $p^i(\bar{\xi})$ is the projected distance of $f^i(\bar{\xi})$ to r^i, and $d(\bar{\xi}) = 0$ constrains $\bar{\xi}$ to M. Finding this maximum value of p^i is possible, since it is just a constrained optimization problem, but it requires having the parameterization of $d(\bar{\xi}) = 0$ and is not worth the effort.

To simplify the process, remove the restriction that $d(\bar{\xi}) = 0$, and try to ensure that c^i lies outside the entire curved face f^i. Let α^i and β^i be disjoint sets of nodes, where the main nodes of r^i are in α^i and the other nodes on f^i are in β^i. For a q_2 element, β^i is just the set of mid-edge nodes on f^i, and for a q_3 element, β^i is the set of the 6 mid-edge nodes and the center node. Also, $p^i_j = 0$ for $j \in \alpha^i$ since the projected distance to a plane of the three points that define that plane is zero. Let:

$$p_{max} = \max(\max_{j\in\beta^i} p^i_j, 0) \tag{11}$$

then p^i can be bounded by:

$$\begin{aligned} p^i &\leq \max_{\bar{\xi}\in f^i} p^i(\bar{\xi}) \\ &= \max_{\bar{\xi}\in f^i} \sum_j p^i_j \phi_j(\bar{\xi}) \\ &\leq p_{max} \max_{\bar{\xi}\in f^i} \sum_{j\in\beta^i} |\phi_j(\bar{\xi})| \end{aligned} \tag{12}$$

for a 2D q_2 triangle:

$$\max_{\bar{\xi}\in f^i} \sum_{j\in\beta^i} |\phi_j(\bar{\xi})| = \frac{4}{3} \tag{13}$$

yielding:

$$p^i(\bar{\xi}) \leq \frac{4}{3} p_{max} = \delta^i \tag{14}$$

This provides a quick way to size the congruent tetrahedron while retaining the property that C contains the entire cutplane intersection, since the bound on δ^i in eq. 14 ensures that c^i will at worst be tangent to f^i.

Now, after the congruent tetrahedron is found for a particular curved element, the standard linear cutplane algorithm is applied to C to determine the shadow primitive. One problem with this approach is that it can sometimes generate a shadow for an element that does not intersect M (e.g. C has one corner clipped by the cutplane). This could be avoided by refining the element and reapplying eq. 7, but these empty shadows are not a problem in practice.

Generating the congruent tetrahedron for a q_3 element is more complicated, but essentially the same process. A face of a q_2 tetrahedron can only be purely concave or convex, while it is possible for the curvature of a q_3 face to have an inflection point or curve. A q_3 face can be classified into one of three groups based on the signs of the p^i_j:

- Mixed: $\exists j,k \in \beta^i$ such that $p^i_j > 0$ and $p^i_k < 0$
- Nonnegative: $p^i_j \geq 0, \quad \forall\, j \in \beta^i$
- Nonpositive: $p^i_j \leq 0, \quad \forall\, j \in \beta^i$

Define for a q_3 face:

$$\begin{aligned} p_{min} &= \min_{j\in\beta^i} p^i_j \\ p^*_{max} &= \max_{j\in\beta^i} \left|p^i_j\right| \\ p'_{max} &= \max(\max_{j\in\beta^i, j\neq 9} p^i_j, 0) \end{aligned} \tag{15}$$

For a mixed q_3 triangle:

$$\max_{\bar{\xi}\in f^i} \sum_{j\in\beta^i} |\phi_j(\bar{\xi})| = \frac{11+160\sqrt{10}}{243} < 2.128 \tag{16}$$

This maximum value occurs at 3 symmetric points, one of which is:

$$\begin{aligned} \bar{\xi} &= \left(\frac{11-2\sqrt{10}}{27}, \frac{11-2\sqrt{10}}{27}, \frac{5+4\sqrt{10}}{27}\right) \\ &\approx (0.173, 0.173, 0.654) \end{aligned} \tag{17}$$

Call a q_3 face nonnegative if all of the $p^i_j \geq 0$. This does not imply that $p^i(\bar{\xi}) > 0$ everywhere, and such a face can be either convex or inflected. For a nonnegative q_3 triangle:

$$\max_{\bar{\xi}\in f^i} \sum_{j\in\beta^i} |\phi_j(\bar{\xi})| = \frac{9}{8} \tag{18}$$

This maximum value occurs at the middle of each edge, that is the 3 points symmetric with:

$$\bar{\xi} = \left(\frac{1}{2}, \frac{1}{2}, 0\right) \tag{19}$$

The bound in eq. 18 can be improved in one special case, when the projected distance of the center node ($j = 9$) is greater than eq. 18 applied to the other nodes in β^i:

$$p^i_9 \geq \frac{9}{8} p'_{max} \tag{20}$$

which implies:

$$p^i(\bar{\xi}) \leq p^i_9 \tag{21}$$

Call a q_3 face nonpositive if all the $p^i_j \leq 0$. Unlike a q_2 face, this face is not necessarily purely concave. Even if no p^i_j is positive, p^i can still extend past r^i. Bounding the maximum value of p^i is a little different for a nonpositive face, since a term $p^i_j\phi_j$ in the interpolation will only be positive if $\phi_j < 0$. Therefore define H to be a step function:

$$H(x) = \begin{cases} 0 & \text{if } x \geq 0 \\ x & \text{if } x < 0 \end{cases} \tag{22}$$

Thus,

$$\begin{aligned} p^i &\leq \max_{\bar{\xi}\in f^i} p^i(\bar{\xi}) \\ &= \max_{\bar{\xi}\in f^i} \sum_{j=0}^{9} p^i_j \phi_j(\bar{\xi}) \\ &\leq \max_{\bar{\xi}\in f^i} \sum_{j\in\beta^i} p_{min} H(\phi_j(\bar{\xi})) \\ &= p_{min} \min_{\bar{\xi}\in f^i} \sum_{j\in\beta^i} H(\phi_j(\bar{\xi})) \end{aligned} \tag{23}$$

For a q_3 face:

$$\min_{\bar{\xi}\in f^i} \sum_{j\in\beta^i} H(\phi_j(\bar{\xi})) = \frac{20-14\sqrt{7}}{27} > -0.632 \tag{24}$$

This minimum value occurs at the 3 points symmetric with:

$$\bar{\xi} = \left(\frac{4-\sqrt{7}}{9}, \frac{4-\sqrt{7}}{9}, \frac{1+2\sqrt{7}}{9}\right) \approx (0.15, 0.15, 0.7) \tag{25}$$

Combining eq. 24 with eqs. 16, 18 and 21 suggests the following logic to compute δ^i for a general q_3 element:

$$\delta^i = \begin{cases} 2.128 p^*_{max} & \text{if } p_{min} < 0,\ p_{max} > 0 & \text{(Mixed)} \\ -0.632 p_{min} & \text{if } p_{min} < 0,\ p_{max} = 0 & \text{(Nonpositive)} \\ 1.125 p_{max} & \text{if } p_{min} \geq 0,\ p^i_9 < p'_{max} & \text{(Nonnegative)} \\ p^i_9 & \text{if } p_{min} \geq 0,\ p^i_9 \geq p'_{max} & \text{(Nonnegative)} \end{cases} \tag{26}$$

The bound for a mixed q_3 element is relatively loose when compared to the bound for a nonnegative element. Also, a purely concave face would be contained by $\delta^i = 0$, as is the case for a q_2 element, but the nonpositive bound in eq. 26 will set δ^i as some positive value. However, q_3 elements are used in a mesh to conform to the curved boundaries of the computational domain, and it is beneficial for the flow solver for these curved boundaries to be well resolved. As a matter of practice, very few of the elements (if any) in a computational grid will be mixed or inflected. In fact, most will be purely concave or convex, and the looser bounds for the mixed elements and nonpositive elements will not be necessary. Assuming that all the elements in a q_3 mesh are either purely concave or convex, this suggests the following logic to compute δ^i for a q_3 element:

$$\delta^i = \begin{cases} 0 & \text{if } p_{min} < 0,\ p_{max} = 0 & \text{(Concave)} \\ 1.125 p_{max} & \text{if } p_{min} \geq 0,\ p^i_9 < p'_{max} & \text{(Convex)} \\ p^i_9 & \text{if } p_{min} \geq 0,\ p^i_9 \geq p'_{max} & \text{(Convex)} \end{cases} \tag{27}$$

6.2 Fragment Shading: Newton-Raphson Inversion

Once the shadow is sent down the graphics pipeline, how are the fragments shaded? Two questions must be answered:

1. Should the fragment be rejected (i.e. is it outside the element)?

2. How is the fragment colored if it is inside the element?

Both of these questions can be answered if the reference coordinates $\bar{\xi}$ of the pixel to be rendered are known. The position is in the element if $\bar{\xi} \geq 0$, and then eq. 2 can be implemented in the fragment shader. The reference coordinates will vary nonlinearly in physical space therefore they can be determined using a Newton-Raphson (NR) inversion algorithm.

At each pixel, the physical coordinates $\bar{x}_p$ are known, since that's what determines the fragment's location via the modelview transformation. For any reference coordinate guess, $\bar{\xi}_i$, the position can be updated using:

$$\bar{\xi}_{i+1} = \bar{\xi}_i + \left.\frac{\partial\bar{\xi}}{\partial\bar{x}}\right|_{\bar{\xi}_i} (\bar{x}_p - \bar{x}(\bar{\xi}_i)) \tag{28}$$

where

$$\frac{\partial\bar{\xi}}{\partial\bar{x}} = \left(\frac{\partial\bar{x}}{\partial\bar{\xi}}\right)^{-1} \quad \text{and} \quad \left.\frac{\partial\bar{x}}{\partial\bar{\xi}}\right|_{\bar{\xi}_i} = \sum_j \bar{x}_j \left.\frac{\partial\bar{\phi}}{\partial\bar{\xi}}\right|_{\bar{\xi}_i} \tag{29}$$

While this is fairly straightforward, the standard OpenGL shading just linearly interpolates color values, so the NR algorithm does represent a significantly larger workload per pixel. However, the only straightforward way to pass nonlinear data through the OpenGL pipeline is through texture maps. Texture maps are are prohibitively expensive to generate for each element, and the additional work of the fragment shader is small by comparison.

6.3 Rendering Results

The shadow method is able to render curved planar cut intersections that are topologically similar to linear cutplane intersections. Fig. 8 shows a triangular cut, where the p_2 tetrahedral element is outlined in black, the congruent tetrahedron is outlined in blue, and the shadow is shown in green and red. Those pixels that are in the cutplane intersection are shaded in green, and the pixels that lie outside the element are shown in red. The figure is shaded to highlight the fact that a linear primitive (the shadow) can be used to render a nonlinear intersection. In a visualization application, the pixels in the shadow outside the element would be discarded by setting their opacity to zero, and the actual intersection would be shaded as in fig. 9. In addition to the two linear cutplane intersections (triangle or quadrilateral), higher order elements can intersect a plane in complicated ways. The shadow algorithm is easily able to capture multiple distinct intersections as shown in fig. 10, and intersections that cut a face without touching an edge as shown in fig. 11.

6.4 Hybrid Selective Refinement

The majority of cutplane intersections will resemble fig. 8, with relatively few pixels in the fragment being discarded. But in examples like fig. 10 and fig. 11, a significant portion of the shadow is eventually thrown away. This extra computational burden can be lessened by using eq. 7 or 9 to selectively refine the element, and then applying the shadow algorithm to each subelement. As shown in figs. 12 through 14, this *hybrid selective refinement* (HSR) algorithm correctly renders the cutplane intersection while requiring much less refinement than LSR would to produce the same level of accuracy.

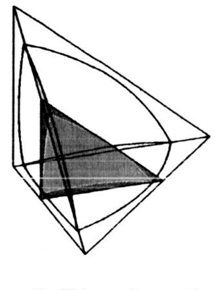


Figure 8: Triangular p_2 Cut w/ Shadow

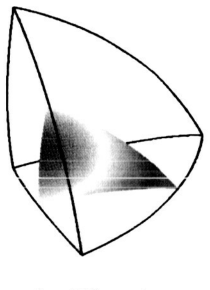


Figure 9: Triangular p_2 Cut Shaded

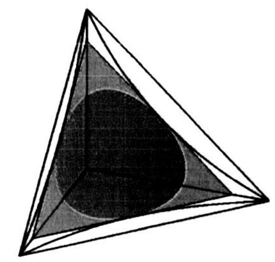


Figure 10: Multiple p_2 Cuts

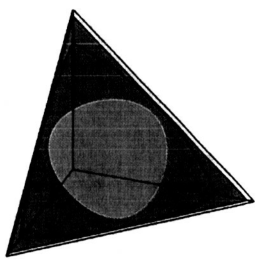


Figure 11: Face Only p_2 Cut

Also notice that there is some amount of overlap between the shadows, but the reduction in excess fragments more than makes up for this redundancy.

6.5 HSR for 2D data

All elements found in the solution from a 2D flow solver can be thought of as occupying a single plane in 3D space. A shadow that lies in that plane can bound the 2D curved element. This shadow primitive will be a linear triangle C that is congruent to the reduced order triangle R of the element, as shown in fig. 15. This is an extension of the method described in sec. 6.4 where the main difference when visualizing 2D data is in computing the bounds of the element. The maximum value of $p^i(\bar{\xi})$ for a q_2 triangle face always lies at the midpoint.

As with sizing the congruent tetrahedron for a 3D tetrahedral q_3 element, the bounds used for a general 2D triangular q_3 element are looser than those actually necessary for elements used in a computational mesh. The bounds for sizing of δ^i for a general element are:

$$\delta^i = \begin{cases} 1.3p^*_{max} & \text{if } p_{min} < 0,\ p_{max} > 0 \quad \text{(Mixed)} \\ -0.316p_{min} & \text{if } p_{min} < 0,\ p_{max} = 0 \quad \text{(Nonpositive)} \\ 1.125p_{max} & \text{if } p_{min} \geq 0 \quad \text{(Nonnegative)} \end{cases} \tag{30}$$

For a q_3 mesh, assuming that the edge is either concave or convex, using:

$$\delta^i = \begin{cases} 0 & \text{if } p_{min} < 0,\ p_{max} = 0 \quad \text{(Concave)} \\ 1.125p_{max} & \text{if } p_{min} \geq 0 \quad \text{(Convex)} \end{cases} \tag{31}$$

will ensure that C completely covers R.

7 Application to Flow Solutions

The method used to intersect finite elements with planar cuts described in previous sections was developed with the goal of visualizing flow solutions on unstructured grids in both 2D and 3D. This

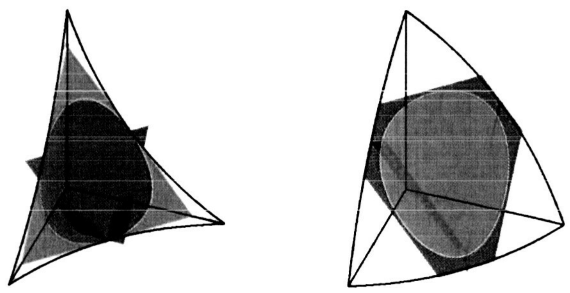


Figure 12: One Hybrid Refinement

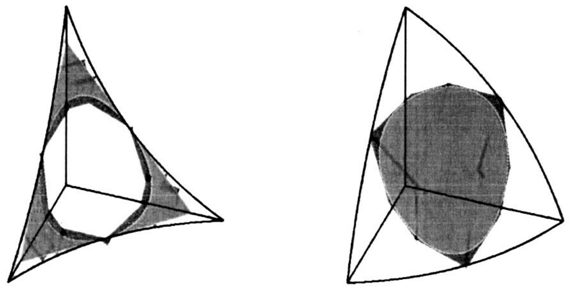


Figure 13: Two Hybrid Refinements

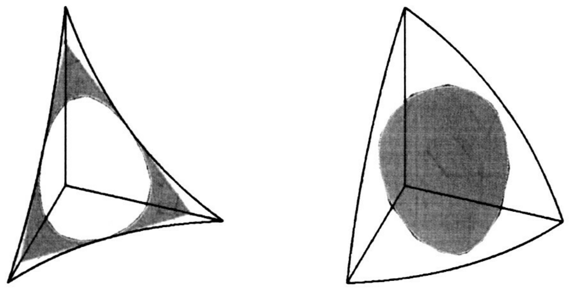


Figure 14: Three Hybrid Refinements

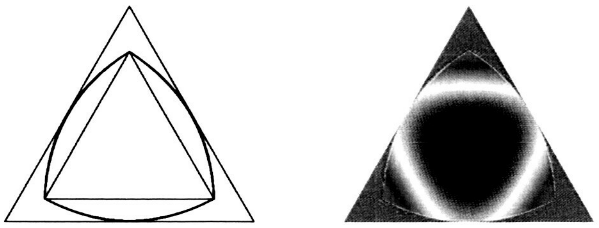


Figure 15: Congruent Shadow Triangle

effort supports the work of Project X [4]. The 2D code solves the Euler equations and the Navier-Stokes equations, while the 3D code is currently only inviscid. The equations are discretized using DG methods and solved using p multigrid with line smoothing.

7.1 2D Viscous Navier-Stokes

The approach to solving the Navier-Stokes equations is the same as the method to solve the Euler equations, except that the line smoothing is modified to account for viscous diffusion in addition to convection. The flow around a NACA0012 airfoil at 0° angle of attack was solved using a grid containing 2264 p_1q_1 triangles in the interior and the farfield, and 40 p_1q_3 triangles on the airfoil. Fig. 16 shows the Mach number distribution, which clearly show both the viscous boundary layer and the trailing wake. Fig. 17 shows a close

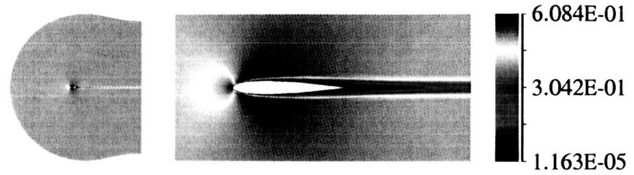


Figure 16: NACA0012 Airfoil Mach Distribution

view of the leading edge, while fig. 18 shows the shadow pixels and outlines the elements. Fig. 19 shows an extreme close-up of just two elements, which are fairly curved. Even at this size, the curvature of the element is preserved.

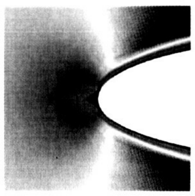


Figure 17: NACA0012 Airfoil Curve

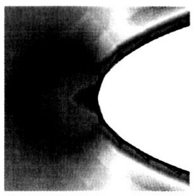


Figure 18: NACA0012 Airfoil Shadows

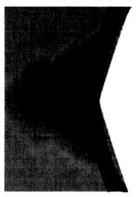


Figure 19: Two Element Shadows

7.2 3D Inviscid Euler

The application of the 3D code is to a straight NACA0012 wing with a span of 5 chord lengths. The grid used was generated from a 2D airfoil grid, which was then extrapolated into 3D. This produced a tetrahedral mesh consisting of 91936 p_2q_1 interior and farfield elements and 3536 p_2q_3 boundary elements around the wing. The Mach Number distribution is shown along the surface of the wing in fig. 20. Since the grid is fairly well refined around the airfoil, no enhancement was necessary to approximate the shape, though the depth and lighting were modified at each pixel in the fragment program to better approximate the curved shape. The farfield boundary forms a dome around the wing, as seen in fig. 21. Fig. 22 also shows the position of the cutplane.

The vast majority of the elements in the grid are q_1, so the standard marching cubes algorithm handles intersection. However, all the elements that either have a face or an edge on the wing surface are q_3, so that they can accurately conform to the airfoil shape. The cuts through these elements were rendered using the shadow method of sec. 6, using eq. 27 to generate the shadows. The curvature at the wingtip is best handled with 1 level of selective refinement, so this was used throughout. The cutplane position in fig. 22 was used to generate the following Mach cut in fig. 23:

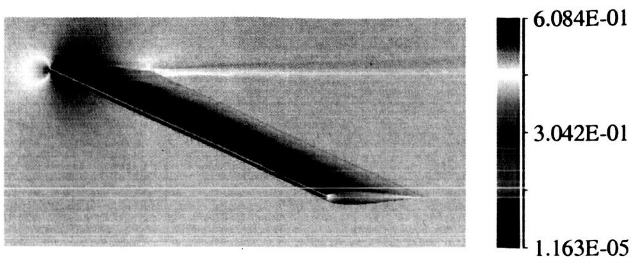


Figure 20: NACA0012 Wing Mach Distribution

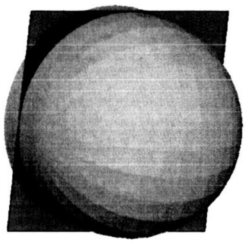


Figure 21: Farfield Boundary

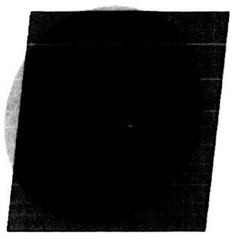


Figure 22: Cutplane Position

To provide a better sense of the element size involved, fig. 24 shows the outline of all the q_3 elements that were cut at the position shown in fig. 22. Figs. 25 and 26 show the cutplane through the leading edge, with all the shadow pixels shown in pink. Notice that there is some overlap of the shadow primitives, but since these pixels normally get rejected, this is never noticed by the viewer.

Fig. 27 shows the wingtip, with the cutplane at 3 locations approaching the tip. These cutplane positions were used to generate images through the Mach field and are displayed in fig. 28. This shows that the cutplane shadow method is able to correctly render the planar intersection for even the fairly curved elements at the wingtip.

8 Extension to Isosurfaces

The discussion so far has focused on rendering planar cut intersections, and not on visualizing isosurfaces. The algorithms to render each type of intersection for linear elements are the same, and indeed, the LSR algorithm should work for isosurfaces. The crucial difference is that isosurface will not, in general, be planar.

However, it may be possible to render the isosurface with scalar value, s^*, by bounding it with linear primitives. Based on screen position, x_s, of each pixel on the bounding shadow, the depth is adjusted until the point on the isosurface, x, is found such that x_s lies on top of x (i.e. x and x_s have the same screen coordinates but different depths). To find x, $|s - s^*|$ is first minimized by performing a search of points inside the element that lie beneath x_s, then

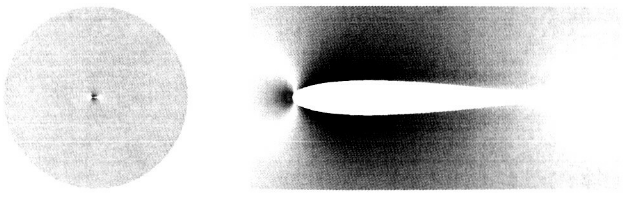


Figure 23: Cutplane Through Mach Field

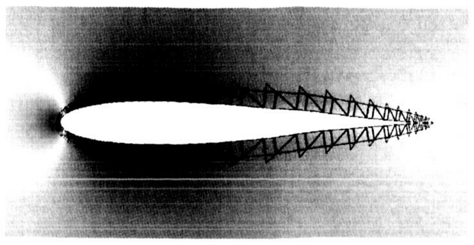


Figure 24: NACA0012 Wing Boundary Elements

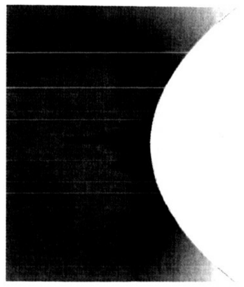


Figure 25: NACA0012 Leading Edge

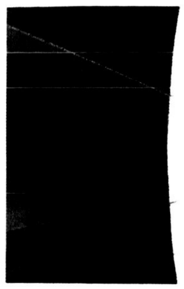


Figure 26: A Few Element Shadows

the fragment can rejected or drawn based on whether or not $s = s^*$. Performing this search would be relatively expensive, so acceptable values of s will lie close to s^* within some bounds set by the accuracy of the search. Under some viewing transforms, the isosurface can curve behind itself, which means there can be multiple solutions, x, that all lie on top of x_s. In this case, the several solutions should be compared using the depth test to determine which one is displayed.

The faces of the congruent tetrahedron used to generate the cutplane shadow would certainly cover the isosurface intersection, since it captures the entire element by design. But using those triangles could produce many extraneous fragments. This could be alleviated by combining the view-based refinement used in [7] and the selective refinement of HSR to approximate the isosurface intersection.

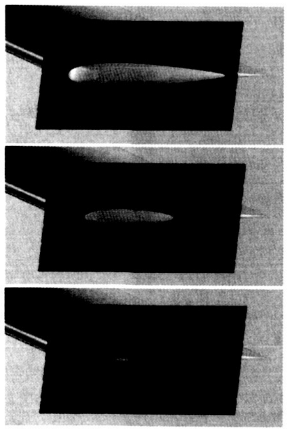


Figure 27: Cutplane Position at Wingtip

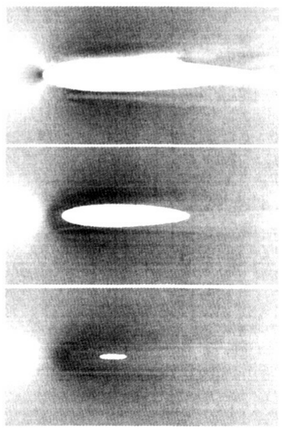


Figure 28: Cutplane Through Mach Field at Wingtip

9 CONCLUSION

Subdivision algorithms generate exponentially more subelements as the refinement level is increased, and their performance is directly tied to the number of vertices being processed. Programmable shaders leverage the flexibility of modern GPUs to efficiently sample higher order data at each pixel in a powerful manner. Visualizing planar cuts through parametric FEM elements simplifies to knowing the reference coordinates at each pixel, and having the ability to use that information to correctly render the scalar field. The major obstacle is the limitation of having to use planar primitives to generate pixels for the fragment shader. To overcome this challenge, the HSR algorithm bounds the curved intersection with a shadow primitive, which can then be manipulated in the GPU. Some pixels will inevitably be discarded, and to minimize this wasted effort, very coarse selective refinement can be used to generate several shadow primitives that collectively cover the entire intersection. Thus the HSR algorithm provides an efficient and functional method to produce and shade planar cuts through higher order FEM data.

ACKNOWLEDGEMENTS

The work presented here was partially funded by NASA grant NAG8-1872 (Suzanne Dorney, technical monitor). Thanks to Krzysztof Fidkowski and Todd Oliver for providing the grids and solutions used in this work.

REFERENCES

[1] James H. Clark. *A fast algorithm for rendering parametric surfaces.* Computer Science Press, Inc., 1988.

[2] Bernardo Cockburn and Chi-Wang Shu. Runge-kutta discontinuous galerkin methods for convection-dominated problems. *Journal of Scientific Computing*, 16(3):173–261, September 2001.

[3] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley, Boston, Massachusetts, 2003.

[4] Krzysztof Fidkowski and David Darmofal. Development of a higher order solver for aerodynamic applications. *42nd AIAA Aerospace Sciences Meeting and Exhibit, AIAA 2004-0436*, 2004.

[5] Mike Giles. Personal Correspondence, July 2003.

[6] B. Haasdonk, M. Ohlberger, M. Rumpf, A. Schmidt, and K. Seibert. Multiresolution visualization of higher order adaptive fi nite element simulations. *Computing*, 70(3):181–204, June 2003.

[7] R. Khardekar and D. Thompson. Rendering higher order fi nite element surfaces in hardware. *Computer graphics and interactive techniques in Austalasia and South East Asia*, 2003.

[8] Andrea O. Leone, Paola Marzano, and Enrico Gobbetti. Discontinuous fi nite element visualization. In *CRS4 Bulletin 1998*. CRS4, Center for Advanced Studies, Research, and Development in Sardinia, Cagliari, Italy, 1998.

[9] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics (Proceedings of SIGGRAPH)*, 21(4):163–169, 1987.

[10] P. Solin, K. Segeth, and I. Dolezel. *Higher-Order Finite Element Methods.* CRCPress, 2003.

[11] Luiz Velho. Simple and effi cient polygonization of implicit surfaces. *J. Graph. Tools*, 1(2):5–24, 1996.

[12] Luiz Velho, Luiz Henrique de Figueiredo, and Jonas Gomes. A unifi ed approach for hierarchical adaptive tesselation of surfaces. *ACM Trans. Graph.*, 18(4):329–360, 1999.

[13] David F. Wiley. *Approximation and Visualization of Scientific Data Using Higher-order Elements.* PhD thesis, University of California, Davis, 6 2003.