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

IEEE Visualization 2004
October 10-15, Austin, Texas, USA
0-7803-8788-0/04/$20.00 ©2004 IEEE

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, $\bar{x}$, are aligned so that the component x_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. $\sum_i x_i = 1$. The nodal shape functions f_i are defined so that at each node n_j:

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

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

$$s(\bar{x}) = \sum_i s_i f_i(\bar{x}) \quad (2)$$

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.

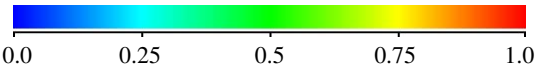

Figure 1: Spectral Colormap

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 $n-1$.

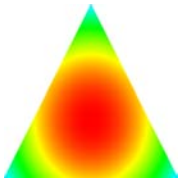

Figure 2: p_2 Shader

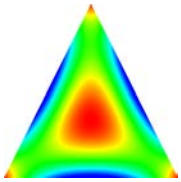

Figure 3: p_3 Shader

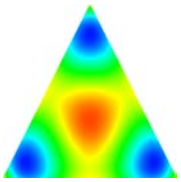

Figure 4: p_4 Shader

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, $\bar{x}$, 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

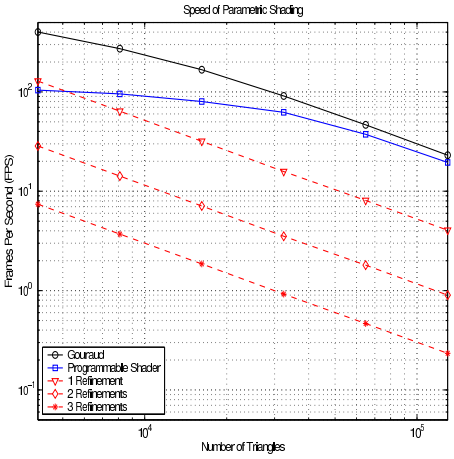

Figure 5: Performance of p_2 Interpolation

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 $\bar{x}$ that satisfy the equation $d(\bar{x}) = 0$.

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:

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

then

$$\begin{aligned} s(\bar{x}) - s^+ &= \sum_i \left(s_i - s^+\right) f_i(\bar{x}) \\ &\leq \sum_i \left(s_{max} - s^+\right) f_i(\bar{x}) \\ &= s^- \sum_i f_i(\bar{x}) \\ &\leq s^- \max_{\bar{x} \in T} \sum_i f_i(\bar{x}) \end{aligned}$$

taking absolute values

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

and noting that for the 3D q_2 shape functions,

$$\max_{\bar{x} \in T} \sum_i \left| f_i(\bar{x}) \right| = 2 \tag{6}$$

which leads to the bounds

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

For the 3D q_3 shape functions,

$$\max_{\bar{x} \in T} \sum_i \left| f_i(\bar{x}) \right| = \frac{16 + 5\sqrt{5}}{9} \leq 3.021 \tag{8}$$

which leads to the bounds

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

The cutplane M will intersect T if $d(\bar{x}) = 0$ at some point $\bar{x}$ 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.

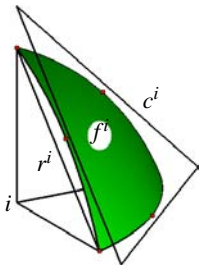

Figure 6: q_2 Orthographic View

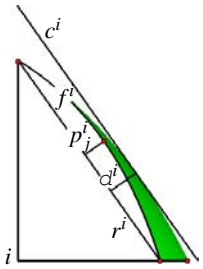

Figure 7: q_2 Side View

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 $\mathcal{d}^i$ that each c^i is offset from each r^i, as shown in fig. 7. If $\mathcal{d}^i \geq 0$, then C always completely contains R. Finding the minimal values of $\mathcal{d}^i$ would require solving:

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

where $p^i(\bar{x})$ is the projected distance of $f^i(\bar{x})$ to r^i, and $d(\bar{x}) = 0$ constrains $\bar{x}$ 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 $d(\bar{x}) = 0$ and is not worth the effort.

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

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

then p^i can be bounded by:

$$\begin{aligned} p^i &\leq \max_{\bar{x} \in f^i} p^i(\bar{x}) \\ &= \max_{\bar{x} \in f^i} \sum_j p^i_j f_j(\bar{x}) \\ &\leq p_{max} \max_{\bar{x} \in f^i} \sum_{j \in \mathcal{b}^i} \left| f_j(\bar{x}) \right| \end{aligned} \tag{12}$$

for a 2D q_2 triangle:

$$\max_{\bar{x} \in f^i} \sum_{j \in \mathcal{b}^i} \left| f_j(\bar{x}) \right| = \frac{4}{3} \tag{13}$$

yielding:

$$p^i(\bar{x}) \leq \frac{4}{3} p_{max} = \mathcal{d}^i \tag{14}$$

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 $\mathcal{d}^i$ 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: $\exists j,k \in \mathcal{b}^i$ such that $p^i_j > 0$ and $p^i_k < 0$
- Nonnegative: $p^i_j \geq 0, \quad \forall\, j \in \mathcal{b}^i$
- Nonpositive: $p^i_j \leq 0, \quad \forall\, j \in \mathcal{b}^i$

Define for a q_3 face:

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

For a mixed q_3 triangle:

$$\max_{\bar{x} \in f^i} \sum_{j \in \mathcal{b}^i} \left| f_j(\bar{x}) \right| = \frac{11 + 160\sqrt{10}}{243} < 2.128 \tag{16}$$

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

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

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

$$\max_{\bar{x} \in f^i} \sum_{j \in \mathcal{b}^i} \left| f_j(\bar{x}) \right| = \frac{9}{8} \tag{18}$$

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

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

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 $\mathcal{b}^i$:

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

which implies:

$$p^i(\bar{x}) \le p^i_9 \tag{21}$$

Call a q_3 face nonpositive if all the $p^i_j \le 0$. 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 $p^i_j f_j$ in the interpolation will only be positive if $f_j < 0$. Therefore define H to be a step function:

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

Thus,

$$\begin{aligned} p^i &\le \max_{\bar{x}\in f^i} p^i(\bar{x}) \\ &= \max_{\bar{x}\in f^i} \sum_{j=0}^{9} p^i_j f_j(\bar{x}) \\ &\le \max_{\bar{x}\in f^i} \sum_{j\in b^i} p_{min} H\left(f_j(\bar{x})\right) \\ &= p_{min} \min_{\bar{x}\in f^i} \sum_{j\in b^i} H\left(f_j(\bar{x})\right) \end{aligned} \tag{23}$$

For a q_3 face:

$$\min_{\bar{x}\in f^i} \sum_{j\in b^i} H\left(f_j(\bar{x})\right) = \frac{20-14\sqrt{7}}{27} > -0.632 \tag{24}$$

This minimum value occurs at the 3 points symmetric with:

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

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

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

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 $d^i = 0$, as is the case for a q_2 element, but the nonpositive bound in eq. 26 will set d^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 d^i for a q_3 element:

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

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 $\bar{x}$ of the pixel to be rendered are known. The position is in the element if $\bar{x} \ge 0$, 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, $\bar{x}_i$, the position can be updated using:

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

where

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

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.

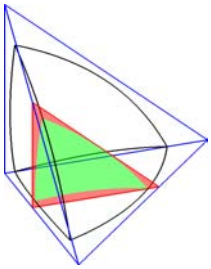

Figure 8: Triangular p_2 Cut w/ Shadow

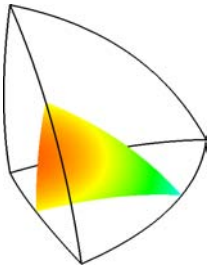

Figure 9: Triangular p_2 Cut Shaded

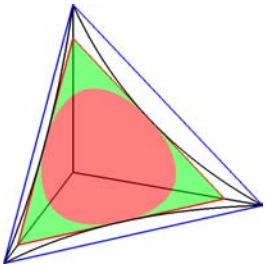

Figure 10: Multiple p_2 Cuts

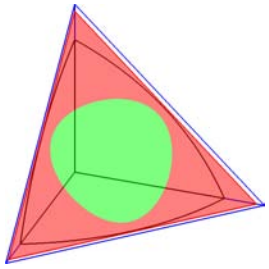

Figure 11: Face Only p_2 Cut

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 $p^i(\bar{x})$ 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 d^i for a general element are:

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

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

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

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

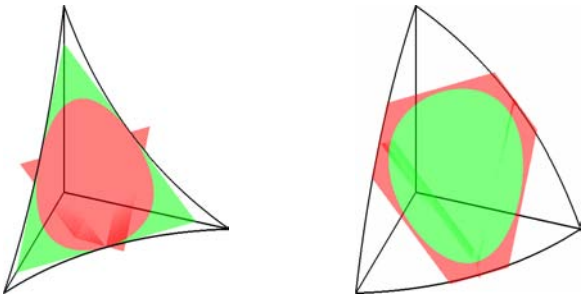

Figure 12: One Hybrid Refinement

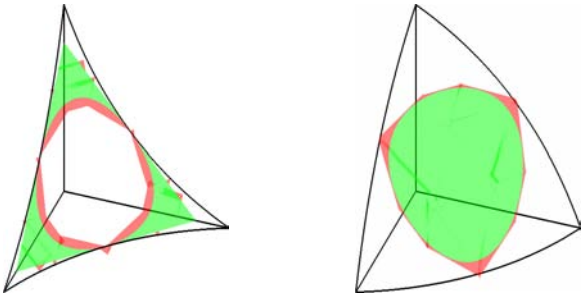

Figure 13: Two Hybrid Refinements

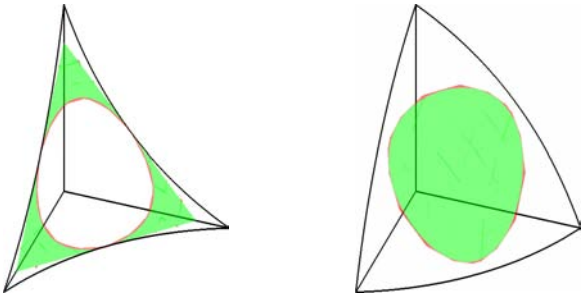

Figure 14: Three Hybrid Refinements

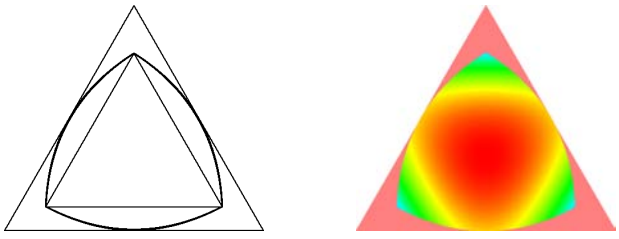

Figure 15: Congruent Shadow Triangle

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

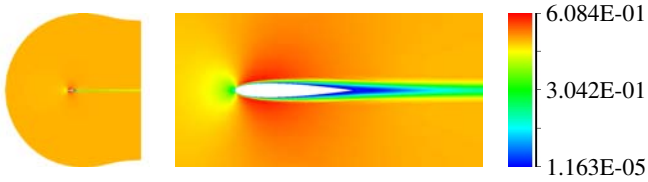

Figure 16: NACA0012 Airfoil Mach Distribution

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.

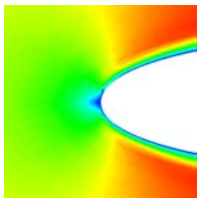

Figure 17: NACA0012 Airfoil Curve

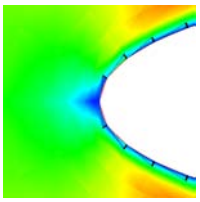

Figure 18: NACA0012 Airfoil Shadows

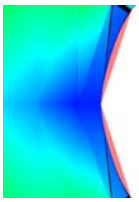

Figure 19: Two Element Shadows

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:

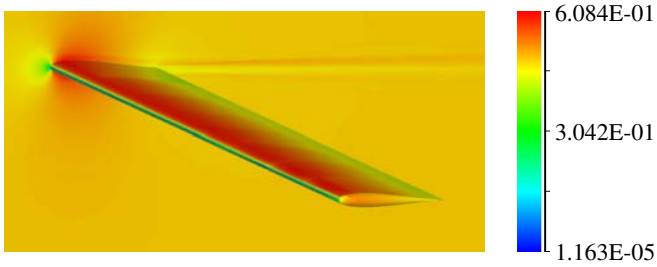

Figure 20: NACA0012 Wing Mach Distribution

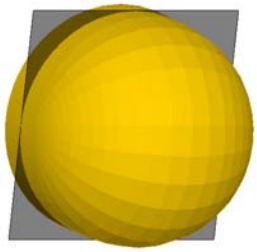

Figure 21: Farfield Boundary

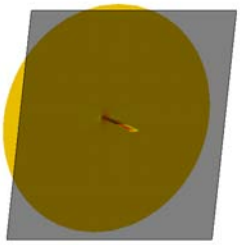

Figure 22: Cutplane Position

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

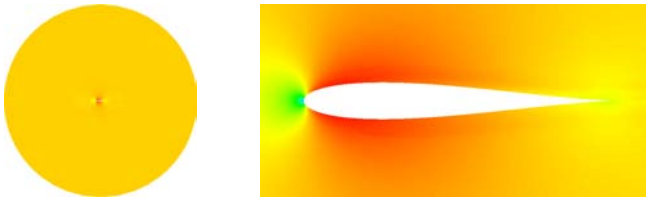

Figure 23: Cutplane Through Mach Field

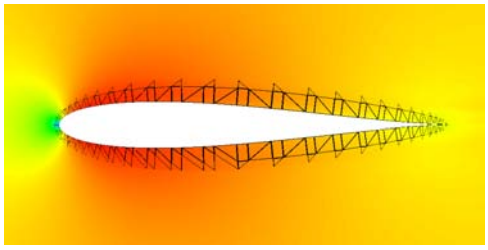

Figure 24: NACA0012 Wing Boundary Elements

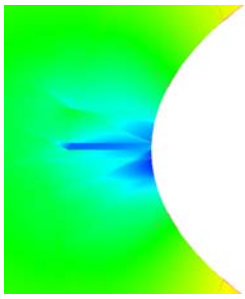

Figure 25: NACA0012 Leading Edge

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.

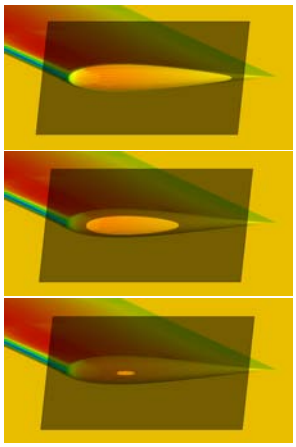

Figure 27: Cutplane Position at Wingtip

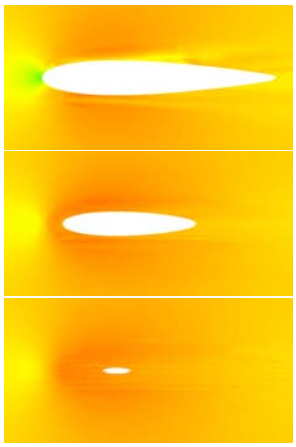

Figure 28: Cutplane Through Mach Field at Wingtip

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 finite element simulations. *Computing*, 70(3):181–204, June 2003.

[7] R. Khardekar and D. Thompson. Rendering higher order finite 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 finite 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 efficient polygonization of implicit surfaces. *J. Graph. Tools*, 1(2):5–24, 1996.

[12] Luiz Velho, Luiz Henrique de Figueiredo, and Jonas Gomes. A unified 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.