# ADVANCEMENT OF COMPUTATIONAL FLUID DYNAMICS VISUALIZATION FOR FINITE ELEMENT APPLICATIONS

By

CODY WAYNE PINKERMAN

Bachelor of Science

Mechanical and Aerospace Engineering

Oklahoma State University

Stillwater, Oklahoma

2008

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 2010

ADVANCEMENT OF COMPUTATIONAL FLUID

DYNAMICS VISUALIZATION FOR

FINITE ELEMENT APPLICATIONS

Thesis Approved:

Dr. A. S. Arena Jr.

Thesis Adviser

Dr. J. Jacob

Dr. J. Conner

Dr. M. E. Payton

Dean of the Graduate College

ACKNOWLEDGMENTS

I want to take a moment and thank several people for their contribution to this thesis and their support in my life over this journey.

To my close friends Ben, Alicia, and Renee.  You three were always there for me as I struggled through this process.  You each took interest in my work, even when you had little to no clue of what was going on.  And you each would stick with me when I needed to rant to release the stress at the times when something went wrong.  You have each enriched my life and been a blessing to me.  Thank you for all the encouragement and your friendship.

To my fellow CASE Lab members Nic and Charles.  You are both brilliant and knowledgeable engineers and were very patient as I began to learn the workings of the CASE Lab.  Nic, you were always there ready to help no matter what the problem; whether it was helping with calculations, proof reading, or helping me brainstorm through a situation in VTK where you had no knowledge of the components.

To my church family.  Thank you for all your prayers and words of encouragements since I started my education here at OSU.  You have made a huge impact in my life and helped me to grow spiritually.

To my advisor Dr. Arena.  Thank you for encouraging me to stay for graduate school and for giving me the opportunity to serve in the CASE Lab.

To my other committee members, Dr. Jacob and Dr. Conner.  Thank you both for taking the time to serve on my committee and for your advice over the last two years.

TABLE OF CONTENTS

## LIST OF FIGURES

Figure Page

# CHAPTER I

# INTRODUCTION

This thesis presents the work that has been done in the area of computational fluid dynamics (CFD) visualizations for the CASE Lab at Oklahoma State University (OSU). In this thesis, we will show how a current visualization program was modified with new features, as well as the current progress in developing a new visualization tool.

In this chapter, we will discuss a brief overview of the visualization program that was used in the progress of this thesis as well as the new visualization tools that were examined during the development of the new visualization software. We will also examine the objectives that were set for the completion of this thesis.

## Overview

The CASE Lab (Computational AeroServoElasticity Laboratory) was initially set up to serve in support of NASA Dryden flight testing. The CASE Lab's goal is to provide fast engineering estimates for flight dynamics, stability, controls, and aeroelasticity. The CASE Lab already has a multitude of capabilities: unstructured two and three dimensional mesh models; an inviscid and viscous flow solver; the ability to model subsonic, supersonic, and hypersonic flow speeds; aeroelastic and controls modeling; non-inertial flight dynamic modeling; aircraft stability derivation; prediction of aeroelastic instability.

The software developed in the CASE Lab has been written using Fortran and C++ computer languages. For the purpose of this thesis, all programming code was written in C++.

To visualize the test cases the CASE Lab has been using a program called GLplot3D. GLplot3D was created by Cowan to work with the CASE Lab's FEM, inviscid solver. GLplot3D was in need of upgrades to keep pace with current needs of the CASE Lab and NASA Dryden. Creating the necessary upgrades to GLplot3D was the first goal that was set for this thesis.

A new visualization tool was found called the Visualization Tool Kit (VTK). VTK offered a multitude of different pre-built options along with the ability to create our own functions (Schroeder, 2006).

This master's thesis is to be used as a stepping stone for a later dissertation in CFD visualization. The focus of the dissertation involves creating a remeshing program to work with CFD solvers and the visualization program. GLplot3D will not be able to handle these types of additions. For these additions to be implemented, the entire GLplot3D code would need to be taken apart and reassembled. The time it would take to dissect GLplot3D and increase the capabilities of GLplot3D would be the same, if not more than, the time it would take to build a new visualization program from the ground up.

## Objectives

When the work for this thesis began, a few objectives were noted:

- Update GLplot3D with new variables and new features
- Learn VTK programming methodology
- Create a basic visualization program using VTK to show comparable capabilities to that of GLplot3D
- Make recommendations for the continued progress of a new visualization program

The first objective was to add upgrades to the current GLplot3D code. GLplot3D needed to be upgraded to ensure the current workload in the CASE Lab could be carried out effectively. The second objective was to learn about VTK and how to program using VTK functions. This process would accomplished by working with example codes to understand the structure and code pipeline that is used in VTK. Once the underlying structure is understood, then new codes could be written for further comprehension.

## Stepping Stone for PhD Dissertation

The current VTK visualization program is just an initial step along the path to a new and better visualization program. In the CASE Lab, we have been looking toward future capabilities in CFD visualization. Work is currently being done involving adding engine data in the current solvers, along with the future desire to create more dynamic simulations involving moving geometries. For some of these applications to be fully realized, major changes need to be made with our current visualization software. Thus, in the best interest of furthering the capabilities of the CASE Lab, a new visualization program would be required. Not only would this program need to have many of the capabilities of the current software, but also be built with the mind set of future aspirations.  With building a software from the ground up with the ideas of future applications, the source code would have much of the framework set up for later upgrade to further the work in the CASE Lab.  It is our goal to build this visualization program over the next few years, and add in the frame work not only for engine and geometry remeshing capabilities, but also new ideas that flow into the CASE Lab with new personnel.

The work done throughout this thesis is the initial step into creating this new visualization program.  The work was completed to give us the background and foundation to further the work in the CASE Lab and CFD visualizations.

# CHAPTER II

# PAST CONTRIBUTIONS

In this chapter, we will discuss the CFD solver used in the CASE Lab along with the current visualization program. We will give back ground information of the current capabilities of the current programs as well as expand on the current needed upgrades and future plans for the CASE Lab visualization.

## Euler3D

Euler3D is the finite element, inviscid, CFD solver used in the CASE Lab (Cowan, 2003). Euler3D has the capabilities to model elastics using the transpiration method. This method involves the solver modeling the motion change not by the actual deflection but by the change in the normal vectors (Stephens, 1998). Euler3D has the capability at modeling rigid body motion using calculations in the non-inertial frame. Rotation is tracked using quaternions for greatest accuracy (O'Neill, 2005).

Euler3D reads in a few different file types: geometry file (.g3d), initial conditions (.unk), elastic mode shapes (.vec), control parameters (.con), and non-inertial matrices (.dyn). The geometry file is set in an unformatted binary file that supplies the solver with the overall data needed to construct the geometry. This data consists of scalar variables and data arrays that are set to the number of nodes, elements, segments, or boundary elements. A time slice output file, .un#, can be renamed to become an initial condition file, .unk. This can be done since both files contain the same information which is also in the same format. The .unk file contains the initial flow field conditions at which to start the calculations. Such conditions include pressure, velocity vectors (Vx, Vy, Vz), density, and enthalpy. The elastics file contains the elastic mode matrices as well as the mode shape vectors for the solid surfaces.

Once Euler3D has completed a test case, five file types are produced: time slices (.un#), modal displacements and velocities (xn.dat), and rigid body displacement and velocities (xd.dat), residuals (*.rsd), and loads (*.lds) files. The time or iteration slices are used for unsteady cases and show the same information as in the .unk file.

Recently, we have worked on a variety of test cases that utilized a number of the features in Euler3D. One such test case is the NASA ARES concept. The ARES (Aerial Regional-scale Environmental Survey) was set in a nose dive trajectory within a Martian atmosphere using the dynamic and control input files. The elevator of the ARES was deflected to three and five degree deflections to simulate a pull up maneuver during Martian entry. These deflections were made in the elastics file. Euler3D ran through about five seconds of flight time per run set (flight position, flight speed, and atmospheric conditions). Each new set was updated with new initial conditions that were taken from the previous run set. This process continued until the actual pull up maneuver was nearly completed. Using the rigid body displacement files, we were able to plot the flight path of the ARES during this maneuver. The ARES test case represents the most complex solution used during this thesis, but other cases were tested to develop the understanding and skill set needed to complete tasks in the CASE Lab using Euler3D: steady supersonic compression corner, more complex supersonic cases, analysis of a Wagner airfoil unsteady time progression, rocket testing during development stage, engine inlet testing during development stage, a simple missile finner, and aeroelastic wing model.

## GLplot3D

Cowan (2003) created GLplot3D as a visualization tool to read in geometry and solution data from the Euler3D program. GLplot3D was designed to read in .g3d file types and its corresponding .unk and .un# data files for steady and unsteady test cases. One feature of GLplot3D is the ability to plot the geometry. GLplot3D can plot the nodes, segments, boundary elements and surfaces. GLplot3D can also plot the properties that are read from the solution files onto the displayed geometry. This includes pressure coefficient, density, Mach number, entropy, and velocity vectors. Using the data from the elastics file, GLplot3D can also show the modal displacements of the geometry.

GLplot3D also gives the user many different options when plotting test cases. The user can turn on and off specific surfaces, choose which variable to display at any given time, show nodes, singular nodes, segments, and wall normals. GLplot3D also can display node numbers for debugging applications.

## The Purpose of the GLplot3D Upgrades

Since GLplot3D was originally created for a course project, many additional applications were left unfinished. Some of the frame work was however set up for later upgrades to be made. That is, many of the user interfaces were created during the initial build to allow later additional programming to be constructed to use these interfaces.

Such interfaces included adding particle traces or stream lines; user specified minimum and maximum scale limits. Some of these upgrades were becoming necessary for the continuation of the work done in the CASE Lab.

## Min and Max

One such interface was for the ability for a user to specify the maximum and minimum limits of the variable side bar and color scheme. This feature would allow the user to manually input a maximum and/or minimum value into GLplot3D and have the visualization change to fit the new range. This ability would be useful for a number of reasons; such as allowing multiple test cases to be easily compared by using the same scale; allowing for specific areas to be more closely examined. With the minimum and maximum limits changeable, a specific area could be “zoomed” in to show a wider color distribution.

## New Variables

As previously stated, GLplot3D originally had five variables that were able to be plotted: pressure coefficient, density, Mach number, entropy, and velocity vectors. GLplot3D reads in data from the .unk and .un# files. GLplot3D reads in the pressure, density, entropy, and velocity data for use not only in plotting, but in calculation of the other variables. Since these are the only values that are being passed from the data files to GLplot3D, any new variables that would be created would need to be based off the original stored values. This course of action would be the best since it would only entail adding to the current GLplot3D code and not changing the solver source code. Velocity magnitude, total energy, internal energy, and vorticity were to be added to the displayable variables in GLplot3D.

GLplot3D currently has the ability to plot the actual velocity vectors for the test case. However, the number of vectors that are created can sometimes be overwhelming and become a hindrance when viewing the solution (Figure 2:1). Thus, there would now be an option when viewing the velocity.

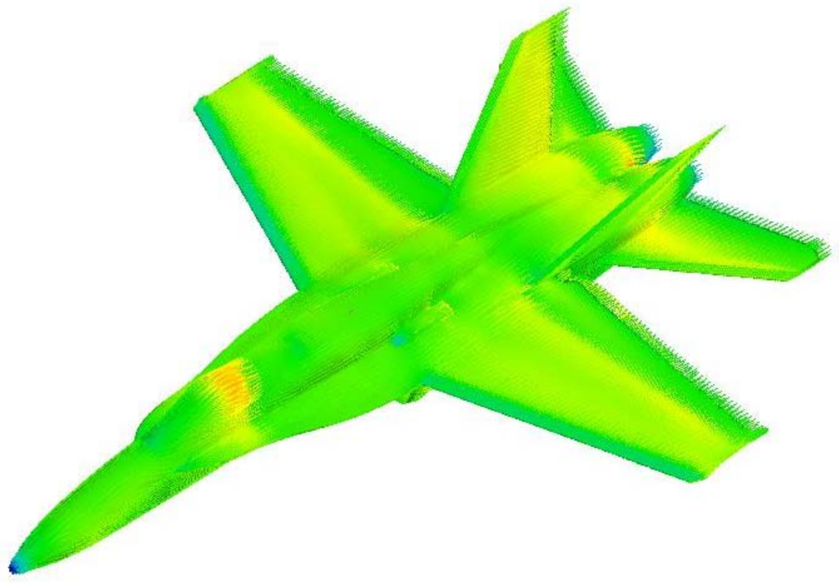


**Figure 2:1: F18 with velocity vectors**

Currently there was no way to be able to discern temperature data, thus where the internal energy addition we would be able to view temperature information. The internal energy calculations are based on the total energy calculations which can be rearranged to be based of temperature instead of pressure values.

## The Purpose of Building New Visualization

The CASE Lab is consistently working towards future expansion in CFD applications. Some of the current projects include adding viscous and turbulence variables, propulsion modeling, higher order elements, and adding the ability for moving meshes. In order to complete these tasks, improvements need to be done with the CASE Lab's current visualization process. GLplot3D, however, does not have the ability to expand to encompass all of the new features. For starters, some of the newer applications will also require the use of higher-order elements. GLplot3D is based on a linear element model and will not be able to handle higher order calculations as well. Also, GLplot3D was designed for the specific purpose to work with Euler3D. This means that for many of the new features to be added, the entire GLplot3D system will need to be deconstructed to understand all of the inner workings, and reassembled to be able to better handle the new add-ons. The time required to change GLplot3D would be similar to the time needed to create a new visualization program from scratch. The decision was made to create a new visualization to have some of the same features as GLplot3D, but to account for future applications.

### VTK

Since a new visualization program would be created, we needed to find a suitable graphics package to for the program. The Visualization Tool Kit (VTK) is an open-source programming tool for computer graphics and visualization (Schroeder, 2006). VTK is a multi-platform graphics tool that contains over 1000 separate class types for use in programming. With an extensive list of callbacks, plotting function, and visualization options, VTK also has the ability to utilize higher-order elements (Schroeder, 2006). This is a huge advantage over GLplot3D, which is based on an OpenGL tool kit. VTK is also capable of being modified to support the needs of the programmer. This opens up a large range of possibilities for a new visualization program as new needs arise.

# CHAPTER III

# THEORY AND IMPLEMENTATION

## Theory Behind Upgrades

In this chapter we will look at the requirements for the GLplot3D upgrades and the code that was generated for the upgrades. The calculations needed to perform the variable upgrades will be discussed to show the reason for the upgrade as well as to allude to the process that will be required for the new code. The process and implementation of the VTK visualization program will also be discussed later in this chapter. We will discuss the overall vision that we will require the VTK program to later achieve, as well as give more background into the architecture of the VTK toolkit. The multiple stages that the VTK code went through will be explained.

### Velocity Magnitude

Each element contains three components of velocity (Vx, Vy, Vz). GLplot3D currently used these values for calculations and displaying of the velocity vectors. One of the desired upgrades was to add the velocity magnitude at each point. Velocity magnitude was found by taking the square root of the summation of the squares of the velocity components.

$$VelocityMagnitude = |\vec{v}| = \sqrt{\vec{v}_x^2 + \vec{v}_y^2 + \vec{v}_z^2}$$

Total and internal Energy

The total energy was found by taking an enthalpy variable and subtracting the pressure at the node.

$$TotalEnergy = \rho E = \rho H - p$$

By viewing the total energy of the system, we can visualize the solution of the energy equation.

Internal energy was then found using the previous total energy and the velocity magnitude of the element. This made implementation easy since the value was based on the previous calculated values.

$$InternalE = e = \frac{\rho E}{\rho} - \tfrac{1}{2}|\vec{v}|^2 = \frac{c_v T}{U_\infty{}^2}$$

The internal energy gives us a way to visualize the temperature of the system.

Vorticity

The last of the new variables was vorticity. Vorticity calculations are based on the curl of the velocity vector.

$$Vorticity = \vec{\omega} = curl\vec{V} = \left( \frac{\partial w}{\partial y} - \frac{\partial v}{\partial z}, \frac{\partial u}{\partial z} - \frac{\partial w}{\partial x}, \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \right)$$

$$|\vec{\omega}| = \sqrt{\left( \frac{\partial w}{\partial y} - \frac{\partial v}{\partial z} \right)^2 + \left( \frac{\partial u}{\partial z} - \frac{\partial w}{\partial x} \right)^2 + \left( \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \right)^2}$$

The curl of the velocity vectors made the vorticity calculations the hardest part of the upgrades to implement. Recalling that each element is composed of four nodes, we were required to find the velocity gradients. The data at the nodes of each element were used to find the volume of the element and shape functions. The shape function, $\Phi_e$, is needed to find the Jacobian matrix which will be used for the partial derivatives.

$$p = \Phi_e \cdot p_e = \begin{Bmatrix} \xi_1 & \xi_2 & \xi_3 & \xi_4 \end{Bmatrix} \cdot \begin{Bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{Bmatrix}$$

Where $\xi_4 = 1 - \xi_1 - \xi_2 - \xi_3$

In coordinate form:

$$\begin{Bmatrix} x \\ y \\ z \end{Bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \end{bmatrix} \begin{Bmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \\ \xi_4 \end{Bmatrix} = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \\ z_1 & z_2 & z_3 & z_4 \end{bmatrix} \begin{Bmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \\ 1 - \xi_1 - \xi_2 - \xi_3 \end{Bmatrix}$$

The Jacobian matrix can now be constructed.

$$J = \frac{\partial \vec{x}}{\partial \vec{\xi}} = \begin{bmatrix} x_{14} & x_{24} & x_{34} \\ y_{14} & y_{24} & y_{34} \\ z_{14} & z_{24} & z_{34} \end{bmatrix}$$

$$x_{ij} = x_i - x_j$$

By finding the Jacobian matrix, the volume of the element could then be found.

$$6\forall = |J|$$

The next step was to find the inverse Jacobian matrix.

$$A = J^{-1} = \frac{1}{6\forall} \begin{bmatrix} y_{23}z_{34} - y_{34}z_{24} & y_{34}z_{14} - y_{14}z_{34} & y_{14}z_{24} - y_{24}z_{14} \\ z_{24}x_{34} - z_{34}x_{24} & z_{34}x_{14} - z_{14}x_{34} & z_{14}x_{24} - z_{24}x^{14} \\ x_{24}y_{34} - x_{34}y_{24} & x_{34}y_{14} - x_{14}y_{34} & x_{14}y_{24} - x_{24}y_{14} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}$$

With the inverse Jacobian, the partial derivatives can be found.

$$\frac{\partial p}{\partial x} = \left( A_{11} \begin{Bmatrix} 1 \\ 0 \\ 0 \\ -1 \end{Bmatrix}^T + A_{12} \begin{Bmatrix} 0 \\ 1 \\ 0 \\ -1 \end{Bmatrix}^T + A_{13} \begin{Bmatrix} 0 \\ 0 \\ 1 \\ -1 \end{Bmatrix}^T \right) p = \begin{Bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \end{Bmatrix} \cdot \begin{Bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \end{Bmatrix}$$

Where $A_{i4} = -A_{i1} - A_{i2} - A_{i3}$

The partial derivatives can be rewritten in a usable format for coding.

$$\frac{\partial u}{\partial x} = A_{11} \cdot u_1 + A_{12} \cdot u_2 + A_{13} \cdot u_3 + A_{14} \cdot u_4$$
$$\frac{\partial u}{\partial y} = A_{21} \cdot u_1 + A_{22} \cdot u_2 + A_{23} \cdot u_3 + A_{24} \cdot u_4$$
$$\frac{\partial u}{\partial z} = A_{31} \cdot u_1 + A_{32} \cdot u_2 + A_{33} \cdot u_3 + A_{34} \cdot u_4$$

This gave the initial way for vorticity to be added to the elements for implementation into the GLplot3D code. The velocities are linearly distributed across the element. Thus the derivatives and the vorticity are constant over a single element. So far, we have analyzed how to add the vorticity over the element, and we need to visualize the vorticity over a node. The initial idea was to give the node the maximum vorticity value that was attached to the node through the different elements that are attached.

$$vort_{node} = Maximum(vort_{node}, vortmag)$$

This however caused incorrect vorticity visualizations. This method resulted towards larger vorticity values so that the entire element with max vorticity appeared red. A single element can dominate a large range of nodes and cause the value distribution to be incorrect.

The second solution was to take the vorticity on a node and add the product of the current volume times the magnitude and then divide the vorticity by the volume.

$$vort_{node} = vort_{node} + vortmag \cdot vol6$$

$$volume_{node} = volume_{node} + vol6$$

$$\omega = {vort_{node}} \big/ {volume_{node}}$$

$$\omega = \frac{\sum_e Volume_e \left|\omega\right|_e}{\sum_e Volume_e}$$

This method showed favorable results, as compared to the first method.

## Upgrading Coding Methodology

In this next section, we will look at the code that was created to implement the previous discussion over the required upgrades to GLplot3D.

### Procedure

The first step taken in order to implement the new upgrades, was for us to become familiar with the coding structure that Cowan used in building GLplot3D. We needed to become familiar with the code to understand the locations for the upgrades, and what systems were already in place for the upgrades.

After sifting through the source code, we found that the input panels for a maximum and minimum interface had already been created in the glik_controls file.

```
glikCreateEditText( "", blk, 18, 200, 60, "override the minimum colorbar value",
NULL );
glikCreateEditText( "", blk, 100, 200, 60, "override the maximum colorbar
value", NULL );
```

These two input panels had not however been initialized to store the values and use them in the color distributing process. The maximum and minimum values used were found by searching for the maximum and minimum data values in the data arrays. These values were then passed to the color assignment parts of the code. This segment of code would then assign color values to the nodes based on the range of the read data. Thus we knew that in order to use the maximum and minimum input feature, we needed to create a new set of variables to overwrite the original minimum and maximum values.

GLplot3D originally had the capabilities of plotting pressure coefficient, density, Mach number, entropy, and velocity vectors. In order to plot the four new variables, we needed new globally defined data arrays. Not only would we need four new arrays for the new variables, but we would also need extra arrays for use in calculating vorticity. The arrays used to find the Jacobian matrix had to be used when reading the geometry data and then were used when reading the solution data. GLplot3D deletes data after reading and performing calculations with the data. Thus for vorticity calculations to be made, we knew we needed to add calculations both during reading in the geometry data and the solution data. The other three variables, velocity magnitude, total energy, internal energy, could be calculated and stored during the reading of the solution data.

## Current Code

Once sample calculations and initial examination of the code was complete, the upgrades were ready to be coded in the GLplot3D software.

<u>Minimum and Maximum:</u> The first step in using the maximum and minimum feature in GLplot3D was to store the data that was typed in the text box into a variable.

This would need to be done in the subroutine with the edit panel controls.  In this subroutine, the codes are executed upon clicking the "OK" button in the interface.  This allowed for the program to inspect the input panels and determine if the user had added values.  Thus, we created two input codes; minimum and maximum.

```
glikRetrieveEditText( edit_glikid[2], 12, &name );
if (name != NULL )
{
        val_min_color = atof(name);
}

glikRetrieveEditText( edit_glikid[2], 13, &name );
if (name != NULL )
{
        val_max_color = atof(name);
}
```

The code would read in the data in the text panels, but would only store it to the appropriate variable if the text panel contained data.  The use of "atof()" allows for the string data stored in the text panel to be used as its numerical representation.

We also wanted to ensure the successful use of this feature in GLplot3D, thus certain scenarios were accounted for.  For example, if the user was to input a maximum value that was smaller than the minimum value.  In this case, a small "if" statement was created to counter this action.  If the user was to perform such a mistake, the code would automatically convert back to the original structure and assign maximum and minimum values based on the values read from the solution data.

```
if (val_max_color <= val_min_color)
       {
          val_max_color=max_vars[current_var];
          val_min_color=min_vars[current_var];
       }
```

The bulk of the minimum and maximum code additions came during the plotting of the variables in GLplot3D. Additions needed to be made to correctly show the values that were read by the minimum and maximum interface. The code segment was put into three parts. The first part ensured that if a node value was larger than the inscribed maximum value, then the node would be colored red. The second segment ensured that any value less than the minimum value was colored purple. The third stage redistributed the color bar to the new scale based on the read minimum and maximum.

```
if (cfd_node[i].val>=max)
        {
                vmd = (float)num_color_divs*1;
                for ( j=0; j<num_color_divs; j++ )
                {
                        if ( vmd <= (float)j + 1.0f )
                        {
                        vmd-=(float)j;
                        Assign_Color_To_Value( (double)vmd, cfd_node[i].color
                        );
                        break;
                        }
                }
        }
        else if (cfd_node[i].val<=min)
                        {
                                vmd = (float)num_color_divs*0;
                                for ( j=0; j<num_color_divs; j++ )
                                {
                                        if ( vmd <= (float)j + 1.0f )
                                        {
                                        vmd-=(float)j;
                                        Assign_Color_To_Value( (double)vmd,
                                        cfd_node[i].color );
                                        break;
                                        }
                                }
                        }
                else vmd = (float)num_color_divs*(cfd_node[i].val - min)/(max -
                min);
                for ( j=0; j<num_color_divs; j++ )
```

```
{
        if ( vmd <= (float)j + 1.0f )
        {
        vmd-=(float)j;
        Assign_Color_To_Value( (double)vmd, cfd_node[i].color
        );
        break;
        }
}
```

Velocity Magnitude: The next upgrade that was coded for GLplot3D was the velocity magnitude. GLplot3D was already capable of reading in velocity data and plotting the velocity vectors, but we wanted to plot just the magnitude. As previously stated, to find the magnitude we just needed to read in the three velocity components, and take the square root of the summation of the squares. Two steps were already implemented in the GLplot3D code. The reading of the velocity data and the summation of the squares of the data had already been implemented.

```
vm2 = cfd_data[0][i].vel[0]*cfd_data[0][i].vel[0]
        + cfd_data[0][i].vel[1]*cfd_data[0][i].vel[1]
        + cfd_data[0][i].vel[2]*cfd_data[0][i].vel[2];
```

The summation of the squares was being used in calculations for other variables at the time. Thus for calculation purposes, we just needed to take the square root of *vm2* and store it.

```
vmd = (float)sqrt( vm2 );
```

Energy: The calculations for total and internal energy both required enthalpy. Enthalpy had been calculated in the Euler3D calculations and written to the data files however, enthalpy was not read into the GLplot3D code. A new global variable was created to grab the enthalpy data from the output files for use in the total energy calculations. This global variable was added in the *cfdvars* array structure (*cfd_data[j][i].ent*) in the *var_decl.h* file of GLplot3D. Enthalpy is the sixth item in the data array that is stored in the Euler3D output files. The data was found in the source files and then used in the energy (*rhoE*) calculations.

```
cfd_data[j][i].ent = (float)vec[i - 1 + 5*np];
…
rhoE = cfd_data[j][i].ent - cfd_data[j][i].pre;
```

Internal energy was then found using the previous calculated total energy and the velocity magnitude of the element.

```
inE = (rhoE/rho)-.5*vmd*vmd;
```

Vorticity: The final coded upgrade was vorticity. After going through and understanding the theory, the way was paved for vorticity to be added into the GLplot3D code. In order for the program to utilize this method, the addition of two new arrays had to be implemented. As stated earlier, to be able to find the partial derivatives, the code needed information about the coordinates of the nodes. In the *cfd_data.c* file, a separate subroutine gathers the needed geometry data and stores it. For the purpose of vorticity, additional arrays were created for storing the A matrix, the four nodes for the element, and the volume. These arrays were created as global variables in the *var_decl.h* file.

```
typedef struct jacob {
        float elnd[4];   //The four nodes for the element
        float inA1[3];  // Inverse jacobian matrix
        float inA2[3];  // Inverse jacobian matrix
        float inA3[3];  // Inverse jacobian matrix
        float Volm6;    // 6*volume
    } jacob;
```

The separate arrays had to be constructed since the geometry data was only used in the one subroutine and then erased, while the velocity data was stored in another subroutine. With the addition of these new geometry arrays, the coordinate data could be read, stored and used in the solution subroutine.

A loop was constructed to perform the geometry input and A matrix calculations. For each element in the mesh structure, the loop would first gather the four nodes, store the information and then gather the nodes' coordinates to calculate the Jacobian.

```
for (i=1; i<=nel; i++)
{
        cfd_elem[i].elnd[0] = iel[i-1];
        cfd_elem[i].elnd[1] = iel[nel+i-1];
        cfd_elem[i].elnd[2] = iel[2*nel+i-1];
        cfd_elem[i].elnd[3] = iel[3*nel+i-1];
x14 = (float)vnd[ne1-1] - (float)vnd[ne4-1];
y14 = (float)vnd[nnd+ne1-1] - (float)vnd[nnd+ne4-1];
z14 = (float)vnd[2*nnd+ne1-1] - (float)vnd[2*nnd+ne4-1];
…
```

The next step was to find the volume of the element and the nine components of the A matrix and store the data for use in the solution subroutine.

```
vol6 = x14*(y24*z34 - z24*y34) + x24*(z14*y34 - y14*z34) + x34*(y14*z24 -
z14*y24);

cfd_elem[i].inA1[0] = (y24*z34 - y34*z24)/vol6;
cfd_elem[i].inA1[1] = (y34*z14 - y14*z34)/vol6;
cfd_elem[i].inA1[2] = (y14*z24 - y24*z14)/vol6;
…
cfd_elem[i].Volm6 = vol6;
```

With the A matrix and volume found and stored, the information could be used in the solution subroutine for calculation of vorticity. The nodes, A matrix, and velocity vectors are called in a separate loop based on the number of elements before the main solution calculations are completed.

```
for (i=1; i<= num_elems; i++)
{
        ne1 = cfd_elem[i].elnd[0];
        …
        A11 = cfd_elem[i].inA1[0];
        A12 = cfd_elem[i].inA1[1];
        A13 = cfd_elem[i].inA1[2];
        A14 = -1*A11 - A12 - A13;
        …
u1 = (float)vec[ne1 - 1 + np];
        u2 = (float)vec[ne2 - 1 + np];
        u3 = (float)vec[ne3 - 1 + np];
        u4 = (float)vec[ne4 - 1 + np];
        …
        dudy = A21*u1 + A22*u2 + A23*u3 + A24*u4;
        dudz = A31*u1 + A32*u2 + A33*u3 + A34*u4;
        …
        vortx = dwdy - dvdz;
        …
        vortmag = (float)sqrt(vortx*vortx + vorty*vorty + vortz*vortz);
```

In the solutions subroutine, the velocity profile is gathered and used for the solutions for each node. For the current vorticity calculation, we had the vorticity magnitude over the entire element. We thus needed to transform the information from the elements to the surrounding nodes.

## Vision on VTK

VTK has the potential to create a sophisticated, vibrant visualization program (Maple, 2006; Kok, 2007; Papademetris, 2009). This visualization software will have all the current capabilities of GLplot3D, but also new applications that are only possible by using the VTK architecture.  The final VTK software will be capable of reading in multiple geometry file types, exporting data on the geometry as well as graphically, showing stream and streak lines, incorporating engine file data, and allowing for the addition of future mesh applications.  We want the new visualization program to be capable of reading in all geometry types that are currently used in the CASE Lab: .g3d, .nc3d, .cfs, etc.  We are also looking ahead at possible file types that may be used in the Mechanical and Aerospace department at OSU in the foreseeable future; ProE, Visual Sketch Pad (VSP) data file types, etc.

The current view is to build a standalone visualization program that will not require outside programs; such as Paraview which is an open-source visualization application program (Squillacote, 2007).  The reason for this is to simplify the complexity of actually running the software, even though this will increase the complexity and length of the code.  But that is a trade off we are currently willing to make for the completion of this program.

## Procedure

The basic architecture of the VTK system is broken into four parts: source, mapper, actor, and renderer (Schroeder, 2006).  The flow of the visualization is the source data which is passed to the mapper; the mapper is then used to create the actor which is passed to the renderer to be displayed.  When first starting to create an object, we need to associate the data needed for the object to the source type structure.  All the data that is needed to create the object, nodes for example, need to be stored into the corresponding dataset. The dataset allow us to define the type of object we are creating. The types correspond to the types of cells being used; whether it be triangles, squares, line, points, pyramids, and many more defined in VTK (Avila, 2006; VTK, 2009). Once the source data has been defined, this data is then passed to the mapper. The mapper is the geometric representation of the source data that is used to define the future actor. Once the mapper is created, the mapper is used to define the actor. The actor is the object that will be drawn with the renderer. The actor is defined by the mapper as well as properties that can be manipulated. Once the actor has been created, it is finally passed to the renderer. The renderer controls the lighting, display windows, cameras, and the displayed object.

Reading the basics of VTK was just the first step in producing a new visualization software. The VTK community had already compiled simple visualization examples for programmers to practice with. Using these examples was the next step. By taking the given examples, simple programs were compile and used for physical examination of the VTK structure.

The first few compiled applications involved a simple 3D cone. The programs would build a cone using hard coded dimensions for height, radius, and resolution and output the structure to the render window. The first compiled cone would rotate about a fixed axis, 360 degrees and then terminate. The second cone example used the same cone from the previous, but added an observer to the code. The observer would output the positions of the cone as it rotated during the program execution. The next cone utilized an interactor, which allowed for the view of the cone to be determined by the user. The mouse interaction would control the rotation, position, and zoom level of the cone in the created window. Other examples involved using multiple render windows, lighting effects, and file types.

Once we had working example codes to compare and experiment with, we decided that to take these working codes and to slowly modify them to fit the needs of the CASE Lab. This was to help ensure that the VTK structure could not only interact with CASE Lab file types, but also ensure that we had a VTK baseline to compare. Modifications to the codes included adding subroutines to read all or parts of CASE Lab file types and use these values in creating the object, to eventually reading in a completely new element and show it in the render window. Once we would be able to create a known geometry structure, we would then evolve the code to be able to read in an entire geometry and output the structure. This entire process would slowly evolve from hard coding the array sizes, file types, and visualized data to be capable of being determined by the software and a user interface.

The CASE Lab's .nc3d file contains all the information needed to describe the geometry of a mesh. This file is constructed using the NetCDF library. Network Common Data Form (NetCDF) is a "set of interfaces for array-oriented data access and a freely-distributed collection of data access libraries for C, Fortran, C++, Java, and other languages." (Rew, 2010) The NetCDF library allows for users to transfer data from one program to another in a compact manner (similar to binary) without worrying about translating between languages. All programs read and write ASCII and NetCDF (instead of binary).

From the .nc3d file, two main segments of data were to be read in by the VTK program: dimensions and variables. The dimensions consisted of scalar values that would determine the length of the arrays of data needed for the geometry, while the variables were composed of an array of data. Dimensions included number of nodes, elements, segments, boundary elements, boundary points, wall nodes, viscous wall nodes, singular nodes, singular viscous nodes, surfaces, and curves. Variables arrays contained data corresponding to coordinates of nodes, nodal data for an element, segment, or boundary element, as well as an index for the boundary conditions of the geometry.

## Initial VTK Experiments

The work done thus far using VTK classes was an evolutionary process. By beginning with example codes for use as a base line, we made initial modifications to understand the class dependencies and uses. From this, we have been able to slowly construct a viable computer program for use in the CASE Lab.

### Cones

Some of the initial examples that were used consisted of a simple cone structure. The first cone program would create a cone, and then rotated the rendered cone 360 degrees. Once this code was successfully compiled, modifications were made. The first modification was to incorporate a subroutine to read in data from one of the many CASE Lab formats. In this case, the .nc3d format was chosen. The original cone code created a cone based off three inputs: cone height, radius, and resolution.

```
vtkConeSource *cone = vtkConeSource::New();
cone->SetHeight( 10.0 );
cone->SetRadius( 5.0 );
cone->SetResolution( 100 );
```

We wanted to create a small subroutine to read in three pieces of data from a .nc3d file, that we would be able to use to create a new cone. Since a .nc3d file is created using the NetCDF library, NetCDF commands were called while reading the data from the file.

A subroutine called "readme" was created using NetCDF commands to be able to read in data from the .nc3d file (Rew, 2009). We needed to be able to first open a file and read in three variables for use in the cone structure. For this initial step, the number of nodes (*nnd*), number of element (*nel*), and number of segments (*nsg*) was read from the .nc3d file and stored into three variables (*r, h, res*).

```
 char filename[256];
 strcpy(filename, "test"); //set filename
 readme(filename); //read in data

void readme(char *name)
{
        int ncid;
        int nndVar, nelVar, nsgVar;

        char filename[256];
        strcpy(filename,name);
        strcat(filename, ".siz");

        ncstatus(nc_open(filename,0,&ncid));

        ncstatus(nc_inq_dimid(ncid, "nnd", &nndVar));
        ncstatus(nc_inq_dimid(ncid, "nel", &nelVar));
        ncstatus(nc_inq_dimid(ncid, "nsg", &nsgVar));

        ncstatus(nc_inq_dimlen(ncid, nndVar, (size_t *) &r));
        ncstatus(nc_inq_dimlen(ncid, nelVar, (size_t *) &h));
        ncstatus(nc_inq_dimlen(ncid, nsgVar, (size_t *) &res));

        ncstatus(nc_close(ncid));
}
```

For the purpose of the first test case, the filename was initially hardcoded to be test.siz. The test.siz file was actually a .nc3d file that was renamed for use in this code. This was to ensure the file was read in correctly and that we knew what was in the file and thus the correct geometry was achieved. Once the file was read, the data stored in r, h, and res were used in the creation of the cone.

```
vtkConeSource *cone = vtkConeSource::New();
cone->SetHeight( h );
cone->SetRadius( r );
cone->SetResolution( res );
```

This basic program eventually was compiled and showed to work as predicted.

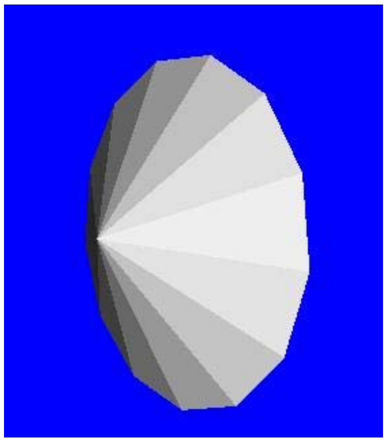


**Figure 3:1: Cone with values read from NetCDF file**

## Alpha

The next phase was to create an actual element to be rendered and capable of user interaction. The first attempt (nc3d_Alpha) would take the NetCDF read commands as previously shown to work in the cone example, but use the data to create a single element and rotate it about an axis as with the cone example.  For us to be able to plot a single element, more data needed to be read in from the .nc3d geometry file: nodes, elements, segments, boundary elements, and surfaces.

```
void readme(char *name)
{
        int ncid;
        int nndVar, nelVar, nsgVar;
        int nbeVar, nsfVar;
        int coorVar;
        int ielmVar, isegVar, ibelVar, lbeVar;

        char filename[256];
        strcpy(filename,name);
        strcat(filename, ".nc3d");

        ncstatus(nc_open(filename,0,&ncid));

        ncstatus(nc_inq_dimid(ncid, "nnd", &nndVar));
        ncstatus(nc_inq_dimid(ncid, "nel", &nelVar));
        ncstatus(nc_inq_dimid(ncid, "nsg", &nsgVar));
        ncstatus(nc_inq_dimid(ncid, "nbe", &nbeVar));

        ncstatus(nc_inq_dimlen(ncid, nndVar, (size_t *) &nnd));
        ncstatus(nc_inq_dimlen(ncid, nelVar, (size_t *) &nel));
        ncstatus(nc_inq_dimlen(ncid, nsgVar, (size_t *) &nsg));
        ncstatus(nc_inq_dimlen(ncid, nbeVar, (size_t *) &nbe));

        ncstatus(nc_inq_varid(ncid, "COOR", &coorVar));
        ncstatus(nc_inq_varid(ncid, "IELM", &ielmVar));
        ncstatus(nc_inq_varid(ncid, "ISEG", &isegVar));
        ncstatus(nc_inq_varid(ncid, "IBEL", &ibelVar));
        ncstatus(nc_inq_varid(ncid, "LBE",  &lbeVar));

        ncstatus(nc_get_var_double(ncid, coorVar, &coor[0][0]));
        ncstatus(nc_get_var_int(ncid, ielmVar, &ielm[0][0]));
        ncstatus(nc_get_var_int(ncid, isegVar, &iseg[0][0]));
        ncstatus(nc_get_var_int(ncid, ibelVar, &ibel[0][0]));
        ncstatus(nc_get_var_int(ncid, lbeVar,  &lbe[0]));

        ncstatus(nc_close(ncid));
}
```

Once the data had been read, it was time to begin allocating the data to the appropriate mappers and actors for rendering.  This required much more than giving the data to an already existing shape function as with the cone.  VTK actors and mappers had to be created to render the element.

The geometries used in the CASE Lab are composed of multiple elements.  These elements are classified as tetrahedral shaped cells in VTK.  This cell class is composed of four points in three dimensional space to create the cell or element.  Not only would we need to draw the cell itself, but eventually would need to show the individual types of boundary elements.  The boundary elements would be composed of specified sides of the elements in the geometry.  Thus both tetrahedral and triangle cell types would need to be used.  Due to this, the actors and mappers were created using an unstructured mesh dataset using the *vtkUnstructuredGrid* function.  This unstructured mesh dataset is capable of representing all possible types of cells in VTK.  Thus one of the first steps was to create an unstructured mesh for the element.

```
//Setup unstructured grid
vtkUnstructuredGrid *element = vtkUnstructuredGrid::New ();  //make grid
element->Allocate (1000);  //do first
```

Now, as mentioned earlier, the element is to be composed of four points in space.  Thus the points or nodes would also need to be defined.  The *vtkPoints* class is capable of creating a data structure for storage of the node and the corresponding location in space and is recognized by *vtkUnstructuredGrid* class.

```
vtkPoints *nodes = vtkPoints::New();
for (j = 0; j<nnd; j++)
{
        //get points x, y, z
        nodes->InsertPoint(j, coor[0][j], coor[1][j], coor[2][j]);
    }
```

When using the *InsertPoint* function, the point number along with the x, y, and z, locations are stored in the allocated memory. The nodal data is stored in the two dimensional *coor* array which is the storage location of the x, y, and z location of the specified node.

*coor[x, y, or z][node id]*

Now that the element structure and nodal information has been created, we needed to link the cells to the corresponding points. *InsertNextCell* is a function that is used by *vtkUnstructuredGrid* to specify the cell type, with the number of points, and a list of the corresponding points. Thus for the creation of a CASE Lab element, we would need to loop through the number of elements that we want to output and link the nodes that make up the specific element.

```
for (int iel = 0; iel < nel; iel++)
{
  List[0][iel] = ielm[0][iel]-1;
   List[1][iel] = ielm[1][iel]-1;
   List[2][iel] = ielm[2][iel]-1;
   List[3][iel] = ielm[3][iel]-1;
   element->InsertNextCell (10, 4, List[iel]);
}
```

With NetCDF, the two dimensional *ielm* array contains information regarding the four nodes that compose the specified element.

*ielm [node 1, 2, 3, or 4] [element id]*

The four nodes are stored in the *List* array which temporarily holds the nodal data for a specific element to be able to pass the data to the unstructured mesh object.  The data in *List* is then replaced by data for next element.  Now that the unstructured grid has been linked to the cell data, we need to link the point data to the element.

```
element->SetPoints (nodes);
```

The *SetPoints* function creates a link the coordinate data that corresponds to the nodes that were previously placed into *element*.

The next step is to use the unstructured grid class and data and create the mappers and actors needed to render the geometry file.  The unstructured mesh data was first linked to a mapper, which was then used to create two separate actors.

```
vtkDataSetMapper *elementMapper = vtkDataSetMapper::New();
   elementMapper->SetInput(element);
   elementMapper->ImmediateModeRenderingOn();

vtkActor *elementActor = vtkActor::New();
   elementActor->SetMapper(elementMapper);
   elementActor->GetProperty()->SetColor(.8,.8,.8); //gray
   elementActor->AddPosition(0,0.001,0);
vtkActor *wireActor = vtkActor::New();
   wireActor->SetMapper(elementMapper);
   wireActor->GetProperty()->SetRepresentationToWireframe();
   wireActor->GetProperty()->SetColor(0,0,0); //black
```

The first actor, *elementActor*, is used to show the actual cell.  This would output the entire tetrahedral with gray colored surfaces.  The second actor, *wireActor*, is to show a wire frame around the boundary surfaces.  For the example of the single element, this will be a wire frame for the entire tetrahedral.

The last step in creating the geometry was to create the actual render window and add the two actors.

```
vtkRenderer *renderer = vtkRenderer::New();
vtkRenderWindow *renWin = vtkRenderWindow::New();
renWin->AddRenderer(renderer);
renderer->AddActor(elementActor);
renderer->AddActor(wireActor);
renderer->SetBackground(1,1,1); //white
```

The first attempt (nc3d_Alpha) was compiled and completed its revolution(s) as designed.

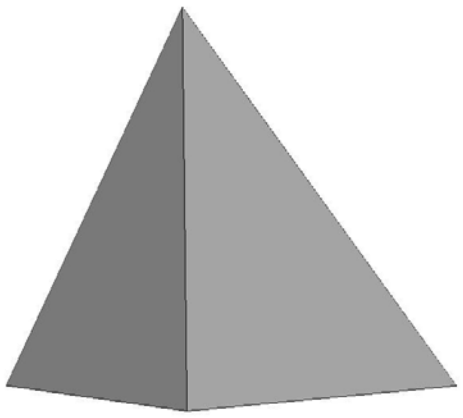


**Figure 3:2: Single element read from .nc3d file that rotated about fixed axis**

Bravo

The next code created (nc3d_Bravo), added the capability of creating a single element and allowing for user interaction to move and rotate the element and not just watch the element rotate about a predetermined axis. The rotation loop was taken out of the original code and replaced with the vtkRenderWindowInteractor function.

```
vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
iren->SetRenderWindow(renWin);
```

This function sets up the interaction between the mouse and the render window.

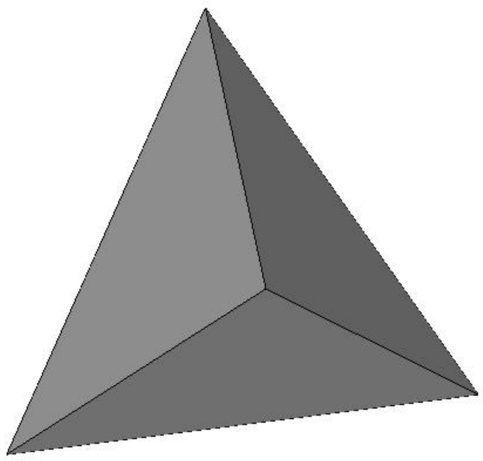


**Figure 3:3: Single element read from .nc3d file with renderer interaction enabled**

Charlie

Once a single element was able to be read into the program and rendered with boundary surfaces and a wireframe, we needed to allow the program to show specific surfaces: such as solid surfaces, symmetric planes, etc. The boundary surface types are read from the .nc3d file and stored in the *lbe* array. The *lbe* array contains eight variables which specify what types of boundaries are attached to the geometry surfaces. For this next step, we focused on solid surfaces. To find what surfaces are classified as a solid surface, the first two variables in the *lbe* array are needed. These two numbers represent the range of boundaries that are classified as solid surfaces. For example, if the numbers one and five were located in the array, then boundary surfaces one through five are considered solids.  This gives us locations that we need to code the solid surfaces.

To add the solid surfaces, we were required to add an additional unstructured grid.

```
vtkUnstructuredGrid *boundary = vtkUnstructuredGrid::New ();  //make grid
boundary->Allocate (1000);  //do first
```

This new structure would first be used to store the solid surfaces.  The solid surfaces are represented by triangular type cells.  Thus the following code was created to determine which three nodes were used for a solid surface and store the data in boundary.

```
for (int ibe = 0; ibe < lbe[1]; ibe++)
{
  sol[0] = ibel[0][ibe]-1;
  sol[1] = ibel[1][ibe]-1;
  sol[2] = ibel[2][ibe]-1;
  boundary->InsertNextCell (5, 3, sol);
}
  boundary->SetPoints (nodes);
```

The *sol* array stores the three nodes that make up the solid surface.

Along with creating a new unstructured grid member, a new mapper and actor would need to be created for this member.  These were set up as before, however in this modification, the triangular surfaces were colored blue to show which surfaces were classified as solid.

```
vtkActor *boundActor = vtkActor::New();
        boundActor->SetMapper(boundaryMapper);
        boundActor->GetProperty()->SetColor(0,0,1); //blue
        boundActor->AddPosition(0,0.001,0);
```

At the end of nc3d_Charlie, the boundary and wireframe actors were the only two actors that were rendered.  This meant that the render would show blue solid surfaces along with a wire mesh around the other boundary surfaces.

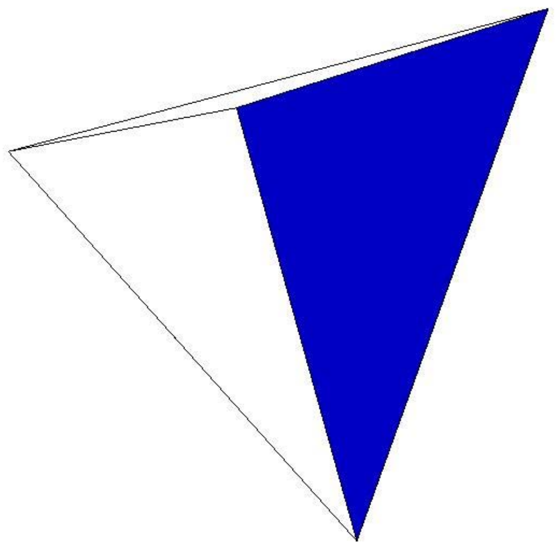


**Figure 3:4: Single element with wireframe and single solid surface**

Delta

The next step was to go from a single element to a multi-elemental object.  This would be the ability to read in a full geometry file and output the entire geometry.  The first step in nc3d_Delta was to create a small geometry and examine the results.  A small box was created that was constructed from seventy-two separate elements.  The first task was to recreate the array sizes for the new object.  At this point, the arrays are being hard-coded to the correct size to ensure proper display.

The arrays needed to be the exact size to function properly. If the arrays were smaller than needed, the program would compile but crash in a runtime error. If the arrays were larger than needed, the nodal placement with the element would be distorted, and an incorrect image would appear.

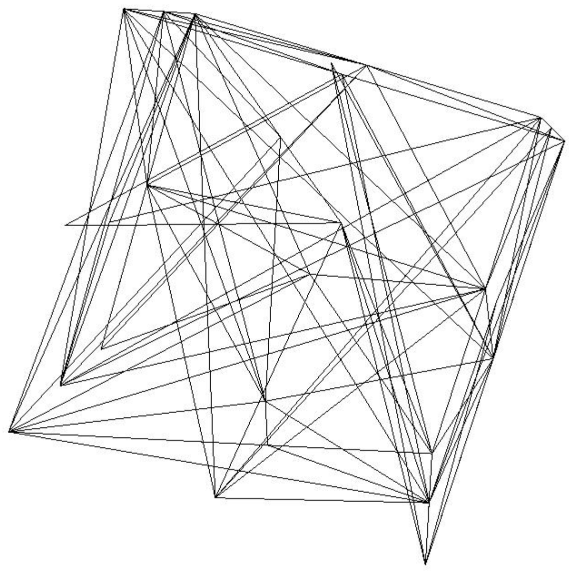


**Figure 3:5: Box wireframe using incorrect array sizes**

The main obstacle at this point was understanding why the arrays had to be the exact size and that the outputted geometry may be incorrect. A system of arrays was created for the box which was hard-coded to correspond to the box's specific values. The values were found using two methods: the values were found during the creation of the geometry file, or by using GLplot3D to determine the correct values.

```
double coor[3][31]; //[3][nnd]
int ielm[4][72]; //[4][nel]
int iseg[2][126]; //[2][nsg]
int ibel[5][48]; //[5][nbe]
int lbe[8];
```

Once again, the boundary and wireframe actors were plotted so that we could visualize the entire domain and the solid surfaces.

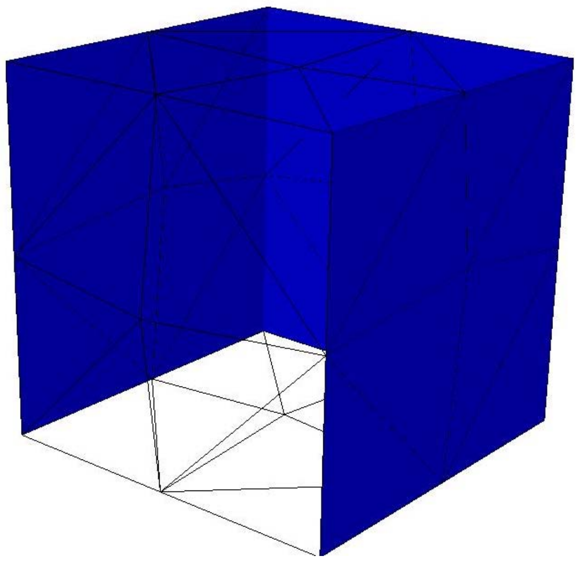


**Figure 3:6: Box geometry showing outer wireframe and three solid surfaces**

The box was a more complex geometry than the single element with multiple surfaces and elements. Thus the next step was to incorporate even more complex geometries to make sure the VTK software could handle large geometry files with thousands if not millions of nodes or elements. In order to test the programs functionality, multiple test cases were hard coded into the program and plotted independently.

```
// Lemon
double coor[3][5663]; //[3][nnd]
int ielm[4][27440]; //[4][nel]
int iseg[2][34797]; //[2][nsg]
int ibel[5][3390]; //[5][nbe]
```

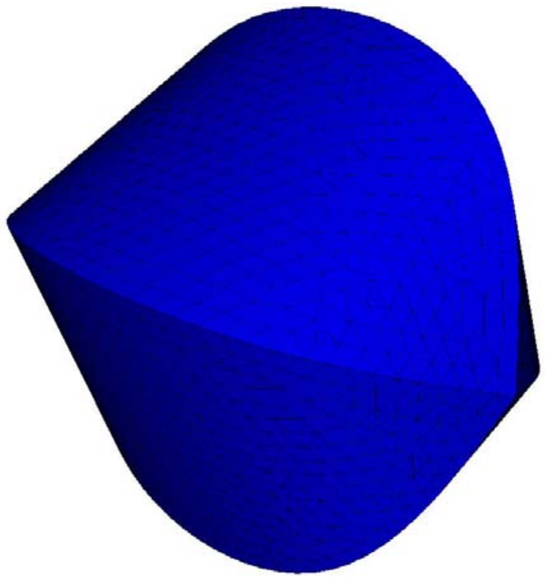


**Figure 3:7: Lemon geometry with wireframe and solid surface**

Echo

The next version was built to incorporate the solution data onto the geometry. We wanted to be capable of reading in the geometry along with a steady solution file (.unk) and plot the color distributions onto the geometry. Reading in data from the .unk file was similar to the geometry. Both file types use NetCDF. For initial testing and coding in nc3d_Echo, only the density data is being used.

```
void readunk(char *name)
{
        int ncid;
        int rhoVar;

        char filename[256];
        strcpy(filename,name);
        strcat(filename, ".unk");

        ncstatus(nc_open(filename,0,&ncid));
        ncstatus(nc_inq_varid(ncid, "Density", &rhoVar));
        ncstatus(nc_get_var_double(ncid, rhoVar,  &density[0]));
        ncstatus(nc_close(ncid));
    }
```

The main purpose of nc3d_Echo is to incorporate the density data and have it plotted on the geometry. In order to do this, the data needs to be transferred to a VTK array type so that it can be stored in the designated structures.

```
vtkFloatArray *newdensity = vtkFloatArray::New ();
for (int inode = 0; inode < nnd; inode++)
{
   newdensity->InsertValue (inode, density[inode]);
}
//input data
node->GetPointData()->SetScalars (newdensity);
element->GetPointData()->SetScalars (newdensity);
boundary->GetPointData()->SetScalars (newdensity);
```

The VTK array structure *newdensity*, is given the node number along with the density data that corresponds to that node. The scalar values are then passed to the actual geometry. This was done three different ways for visualization. First way was to pass the data to just the nodes for visualization, the other was to cover the entire domain with the use of the *element* structure, and the final was to allow plotting on only the solid surfaces using *boundary*.

To visualize and be able to interpret the data, the nodes or surfaces would need to be colored to represent the value of the solution data. The first step to start to create this coloring system was to create reference of what values of color should be used. The range of colors was based off a hue circle.

```
// Now create a lookup table that consists of the full hue circle
  vtkLookupTable *hueLut = vtkLookupTable::New ();
        hueLut->SetTableRange (0, 1);
```

```
hueLut->SetHueRange (.667,0); // 240/360=blue
hueLut->SetSaturationRange (1, 1);
hueLut->SetValueRange (1, 1);
hueLut->Build ();     //effective built
```

The hue coloring scheme is based on three factors: hue, saturation, and value. The hue value determines the range of colors that will be used based on the hue circle. For this case, we wanted to create a color system that would go from blue to red.

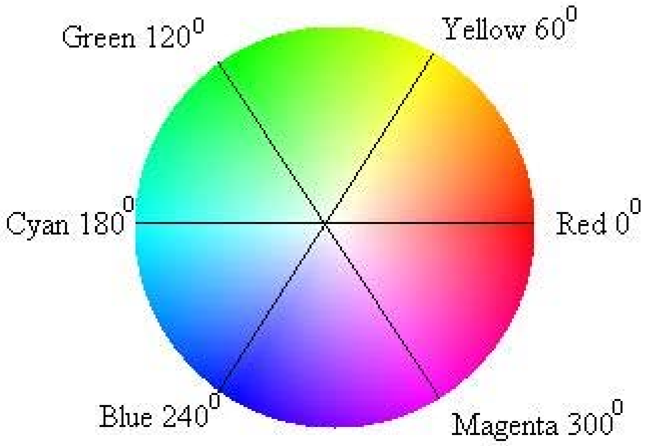


**Figure 3:8: Hue circle**

The saturation value determines the lightness or intensity of color; the range of colors going from white to the actual color. While the color's value determines the darkness or the color: the range of colors going from black to the actual color. Based on this system, we wanted colors that were the actual color on the hue circle and not darken or brightened in any way. Thus the saturation and value levels were set to (1,1).

A new mapper was created to allow the data to be shown using a color scale so that the data could be interpreted. For the initial testing, the solution was only plotted on the solid surfaces.

```
//Color Mapping
vtkDataSetMapper *datamapper = vtkDataSetMapper::New();
        datamapper->SetInput(boundary);
        datamapper->SetScalarRange(boundary->GetScalarRange());
        datamapper->SetLookupTable(hueLut);
```

This section of code links the solution and geometry data that is stored in *boundary* to *datamapper*. This is also where the range of the density data is found for use in the coloring system. Also the previously created *hueLut* which contains the colors to be used is also linked to *datamapper*.

The color scale needed to be created for the user to understand the plotted results. The *ScalarBar* function was used to create a color bar at the side of the render window. This scale would display the range of colors along with the numerical range of the data.

```
//Scalar color bar
vtkScalarBarActor *scalarBar = vtkScalarBarActor::New();
        scalarBar->SetOrientationToVertical();
        scalarBar->SetTitle("Density");
        scalarBar->SetNumberOfLabels(2);
        scalarBar->SetLookupTable(hueLut);
```

The color bar was created to show the minimum and maximum values of the density data as well as show the variable that is being outputted (“Density”). Once again this used the color data that was initially set up in *hueLut*.

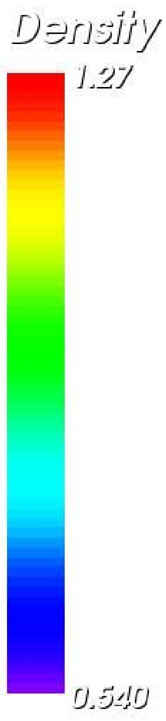


**Figure 3:9: Echo version density scale**

The final few changes made to nc3d_Echo was to create the actor from *datamapper*, along with plotting the new actor and scalar bar in the render window.

```
vtkActor *dataActor = vtkActor::New();
        dataActor->SetMapper(datamapper);
…
renderer->AddActor(scalarBar);
renderer->AddActor(dataActor);
```

Now it was time to test the additions to the code.

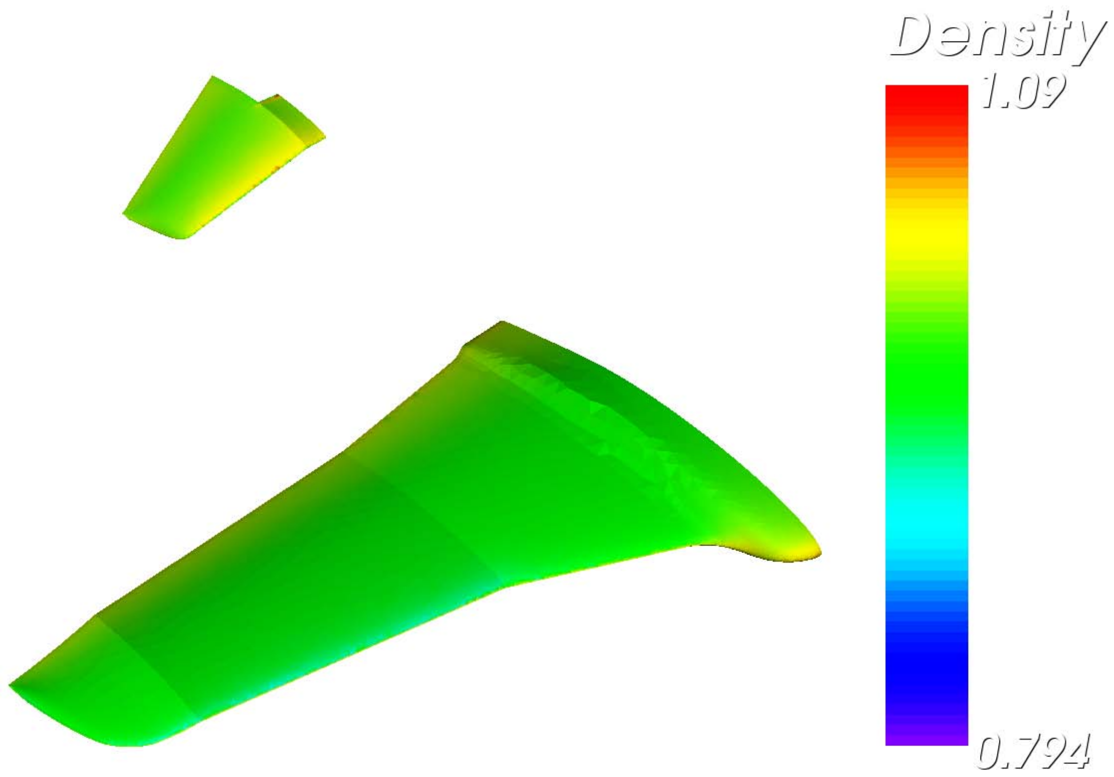


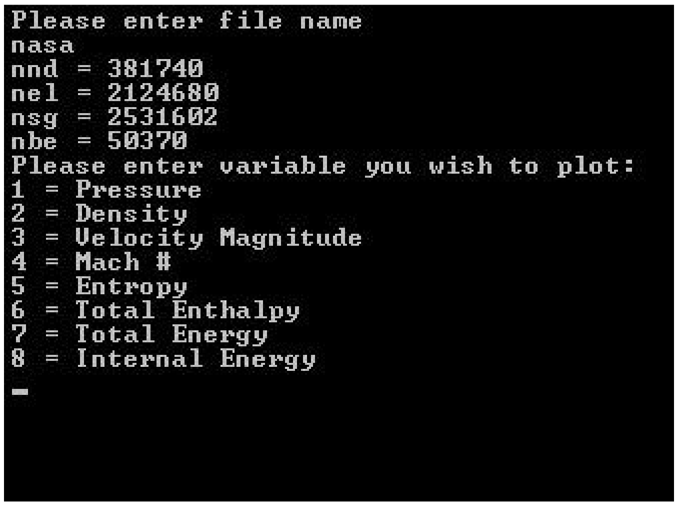
**Figure 3:10: NASA ARES density plot using version Echo**

Foxtrot

As shown, the programming of a VTK visualization program has gone through many stages already and there is still a long road till having a complete package. The current stage, nc3d_Foxtrot, has many features that the CASE Lab can start using in our validation.

Version Foxtrot is a step-up from the version Echo. In Echo, only the density data was read from the solution file. In Foxtrot however, pressure, velocity vectors, and density data is being read into the program. This gave the program access to plot multiple variables. With these five pieces of data, the program could actually calculate up to eight different variables: pressure coefficient, density, velocity magnitude, Mach number, entropy, total enthalpy, total energy, and internal energy.

A small user interface was added into the DOS prompt to allow the user to select which of the eight variables they wanted to plot. This interface currently outputs a number of options for plotting solution data and will only calculate the selected option for viewing. Not only will the render plot the correct variable, but also display the correct title and data labels on the scalar bar.



```
Please enter file name
nasa
nnd = 381740
nel = 2124680
nsg = 2531602
nbe = 50370
Please enter variable you wish to plot:
1 = Pressure
2 = Density
3 = Velocity Magnitude
4 = Mach #
5 = Entropy
6 = Total Enthalpy
7 = Total Energy
8 = Internal Energy
_
```

**Figure 3:11: Foxtrot user interface**

It should be noted that currently, the array sizes used for storing the geometry and solver data are still hardcoded into the program. By using the NetCDF functions, the arrays used for storage must be the exact size as the variable that is being read. For example, if a geometry contains thirty one nodes, then the density array must be thirty one units long for storing the data. If the array is shorter or longer, one of two things will happen. One possible outcome is that the render window will not render and the program will crash.  The second possible outcome will be that the geometry will be plotted incorrectly with some elements containing the wrong node set.

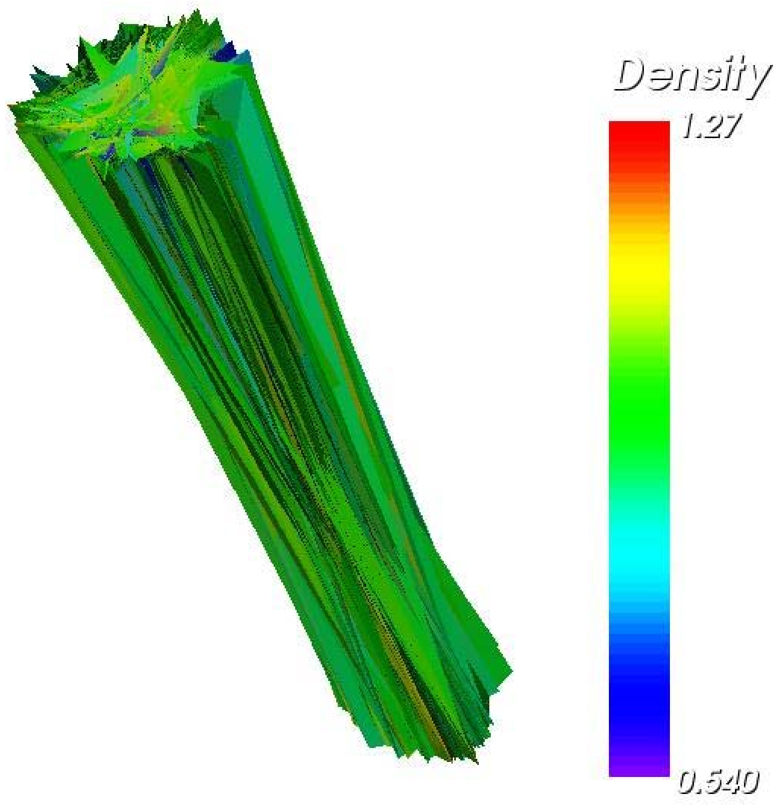


**Figure 3:12: ARES file uploaded using incorrect array sizes**

Another feature that was added was the addition of symmetry planes with the outputted geometry.  Currently, version Foxtrot will display all solid and symmetry surfaces for use in displaying the data.

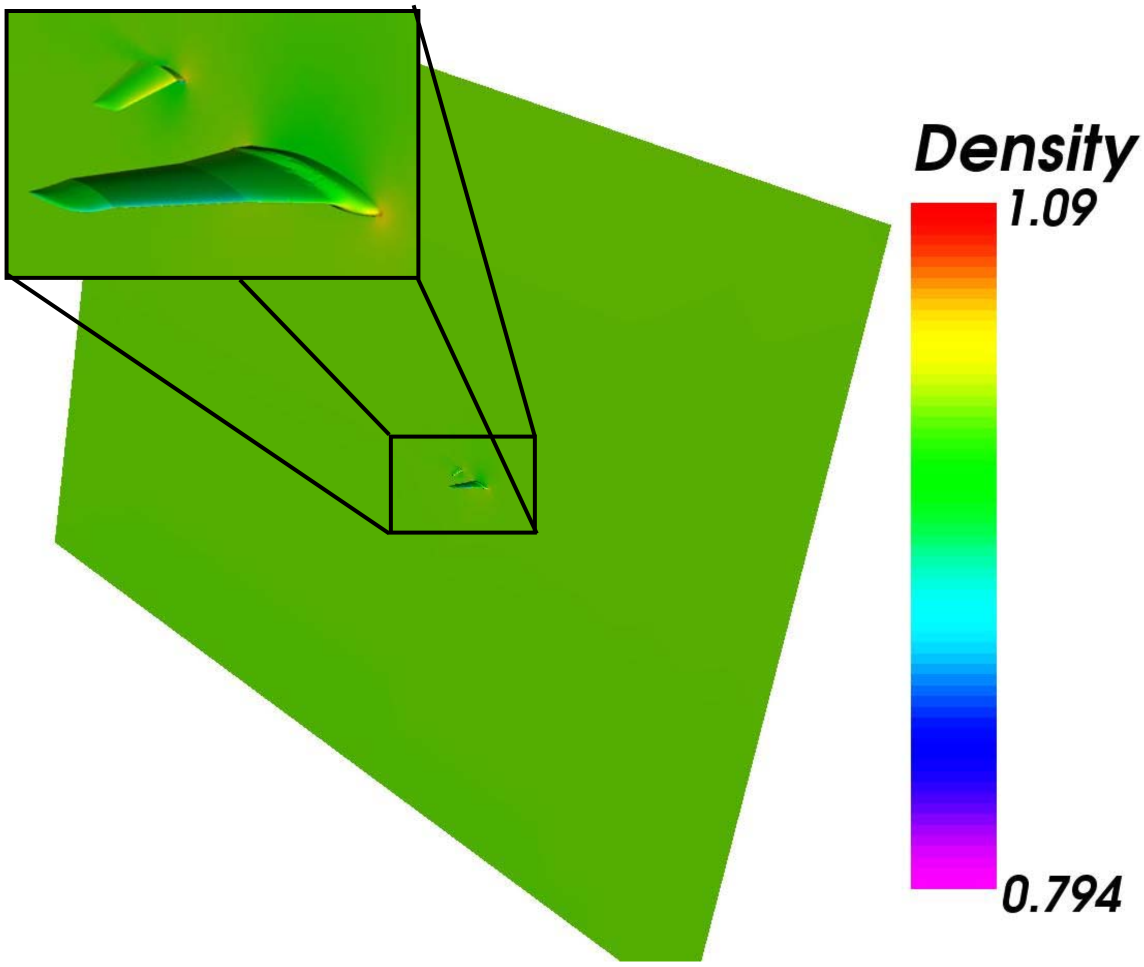


**Figure 3:13: Foxtrot density plot of ARES half mesh with symmetry plane**

More examples from Foxtrot will be shown in the Results chapter to follow.

# CHAPTER IV

# RESULTS

In this chapter, we will recall the test cases that were built and run throughout the course of this thesis, as well as examine the outputs of the GLplot3D upgrades and the final version of the VTK visualization software. For the GLplot3D upgrades, we will see the four new variables on a series of test cases, along with some examples of the minimum and maximum feature that was added. As for the VTK visualization program, we will show some example test cases along with the same case plotted in GLplot3D to compare them and show some of the current features built into the VTK software.

## Test Cases

The following test cases were investigated in this study during development and testing of the two visualization codes:

| Test Case | Geometry Built/Modified By Author | Solution Run by Author | Used for Introduction in CFD | Used for GLplot3D Upgrade Validation | Used for VTK & GLplot3D Comparison |
|---|---|---|---|---|---|
| Box | X | X | X | X | X |
| Supersonic Compression | X | X | X |  | X |
| Upward Facing Wedge | X | X | X |  |  |
| ARES | X | X | X | X | X |
| ARES with Symmetry Plane | X | X | X | X | X |
| 2D Afterburner | X | X | X |  |  |
| 2D Rocket Nozzle | X | X | X | X | X |
| 3D Rocket Nozzle | X | X | X | X | X |
| Turbo Fan | X |  | X | X | X |
| F18 |  |  |  | X | X |
| F22 |  |  |  |  | X |
| Cavity for Tips | X | X | X | X | X |
| Cavity with NACA Airfoil | X |  |  |  |  |
| Aeroelastic Wing | X |  | X |  |  |
| Hypersonic Vehicle |  |  |  | X | X |
| Sphere |  |  |  |  | X |
| Wagner Airfoil |  | X |  | X | X |
| SD 7083 Airfoil |  |  |  | X | X |
| NASA Supercritical Airfoil |  |  |  | X | X |

**Figure 4:1: Test cases used throughout thesis**

### Box

A box geometry was created to help in validation and education of the visualization process in the CASE Lab. The box was a simple geometry that could be remeshed to help in testing the new VTK visualization program when reading more complex geometries. Variable distributions were defined on the geometry to validate updates that were added to GLplot3D.

### Supersonic Compression

The first test case that was run at the beginning of this thesis was a supersonic compression test case. This test case involved a boxed geometry with a small incline to cause a shock. The geometry was already built, so that we could run the test case in order to learn to use Euler3D. This initial test case gave us the process needed to obtain solutions using Euler3D. The test case was first run using a default, coarse mesh, but after analyzing the initial solution, the geometry was remeshed to allow for the shock to be clearly visible.

### Upward Facing Wedge

Another test case, similar to the supersonic compression, involved an upward facing wedge. Once again the geometry had already been created, however the geometry required multiple refined meshes to be capable of clearly showing the multiple shocks and single expansion fan. After a solution finished running, the mesh was reset to allow for better results.

## ARES

This test case was originally constructed by J. Coltrane for modeling stability and performance of the NASA ARES Vehicle (Coltrane, 2005). The geometry was used for the purpose of this thesis to use Euler3D's elastic features. The tail geometry was deflected to multiple deflection angles to examine the effects of the mesh and solution results.

## ARES with Symmetry Plane

The run time for the original ARES mesh was around seven days. In order to minimize the solution time, the geometry was modified. The geometry was cut in half along with the addition of a symmetry plane. Along with changing the default geometry, the total number of nodes was decreased again by changing the meshes that were on the geometry. After these changes, the run time was decreased to one day. The ARES geometry was run to model elastics and dynamics for a pull up maneuver during entry into the Martian atmosphere.

## 2D Afterburner

Propulsion modeling is being added to the capabilities of the CASE Lab. A simple 2D afterburner was created to begin testing the new engine input files. The throat and exit areas of the afterburner geometry were initially created based on the design profile for the afterburner. The overall shape of the converging diverging nozzle was created using cosine curve.

## 2D & 3D Rocket Nozzles

After the initial afterburner geometry was built and tested, we noticed additional shocks and expansions inside the nozzle. This was due to the generalized shape of the nozzle. By using the Method of Characteristics (Emmons, 1958), a new nozzle was built to cancel the formed shocks across the length of the nozzle. This new rocket nozzle was then run through Euler3D for multiple different pressure ratios in order to model an internal normal shock, normal shock at the exit, oblique shocks at the exit, and the optimal design point. From Figure 4:2, we can analyze three different pressure values that were read from the engine file, and visualize the shock as it travels through the nozzle.

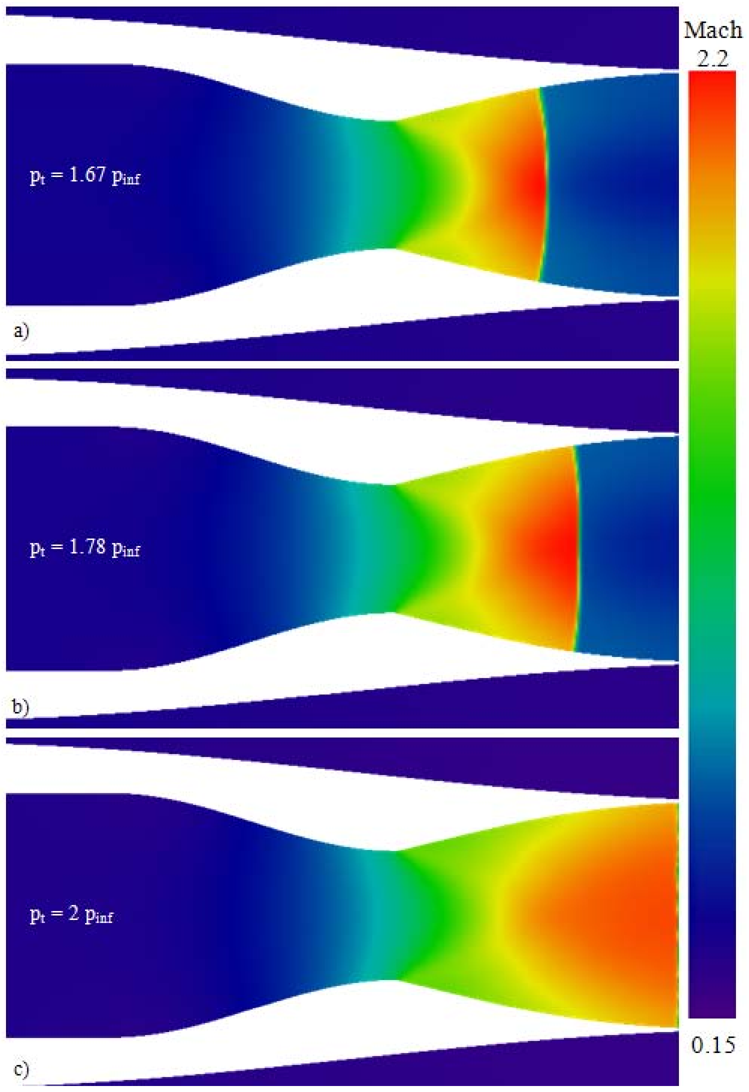


**Figure 4:2: Local Mach number in a rocket nozzle**

## Turbofan

A model of a 2D turbofan was created for more propulsion testing. The upper and lower surfaces were created based on the shape of the top of an airfoil and also included the newly added engine boundary types for the inlet and exit of the turbofan. An afterburner was also added to the exit of the turbofan.

### Cavity Cases

A simple cavity model was created to support current cavity wind tunnel research at OSU. One such cavity was created with two sharp tips that extended into the cavity. This case was run through Euler3D. The geometry was later changed to incorporate an airfoil at the downstream tip.

### Other Test Cases

Other test cases have been created and solved while we have been involved in the CASE Lab. Some of these test cases were converted to the NetCDF format for use in validating the new GLplot3D upgrades and for validating the new VTK visualization software. Other cases included an F18, F22, hypersonic vehicle, NASA supercritical airfoil, SD 7083, sphere, etc.

## GLplot3D Upgrade Results

We first begin by looking at the new features added to GLplot3D: velocity magnitude, total energy, internal energy, and vorticity. The first case that we will show is the NASA ARES model. The velocity magnitude, total energy and internal energy were plotted for this test case. The velocity magnitude figure is shown in comparison with the velocity vectors.

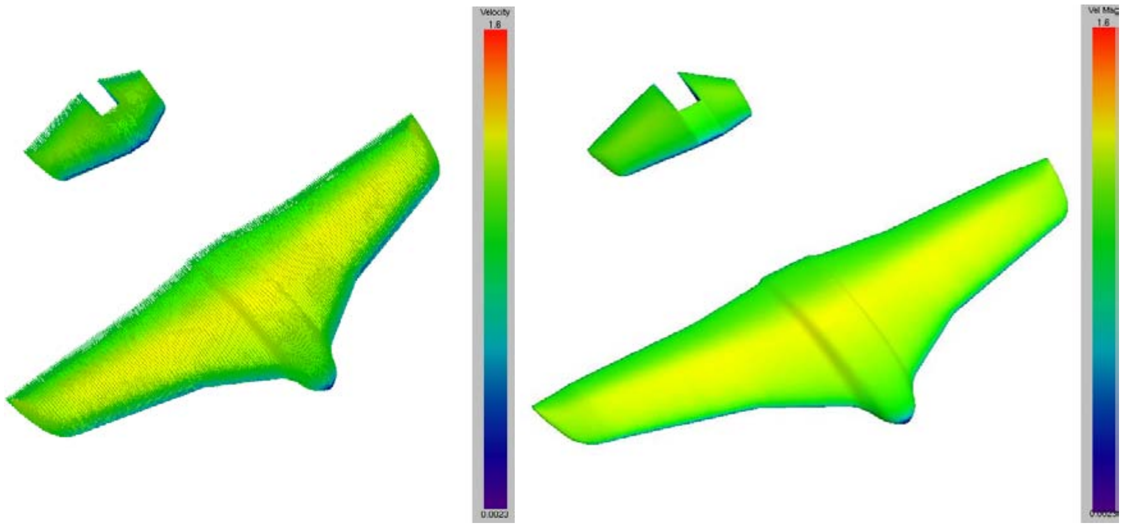


**Figure 4:3: Velocity magnitude vs. velocity vectors for NASA ARES**

For some test case, viewing just the velocity vectors can shift the color distribution. Figure 4:4 shows a comparison of a simple sphere test case, where the velocity distribution is shifted.

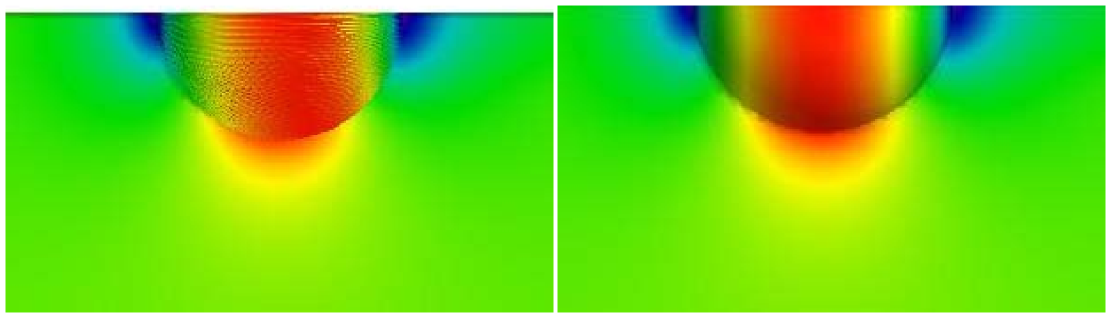


**Figure 4:4: Velocity comparison over a sphere (vectors left, magnitude right)**

The next new variable was total energy. The total energy is shown for a series of test cases.

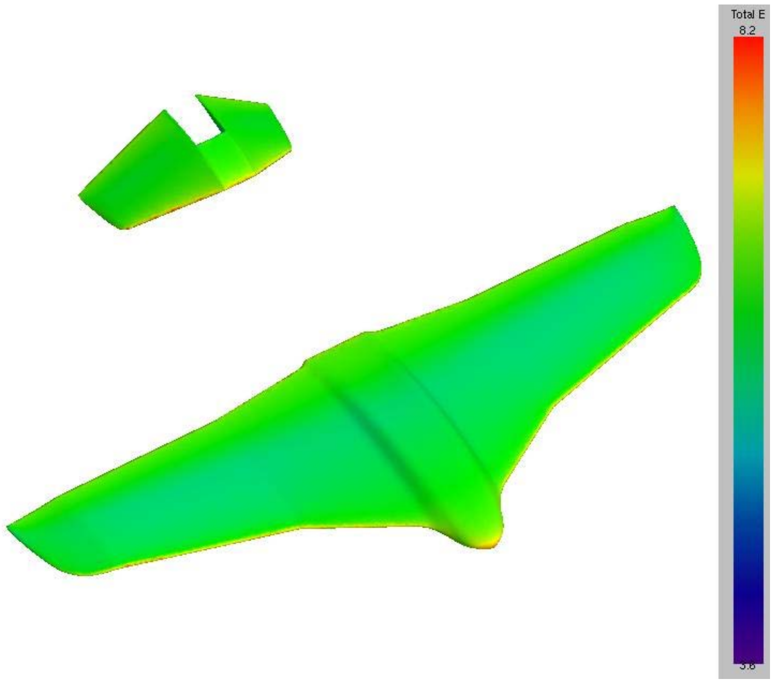


**Figure 4:5: Total energy for NASA ARES**

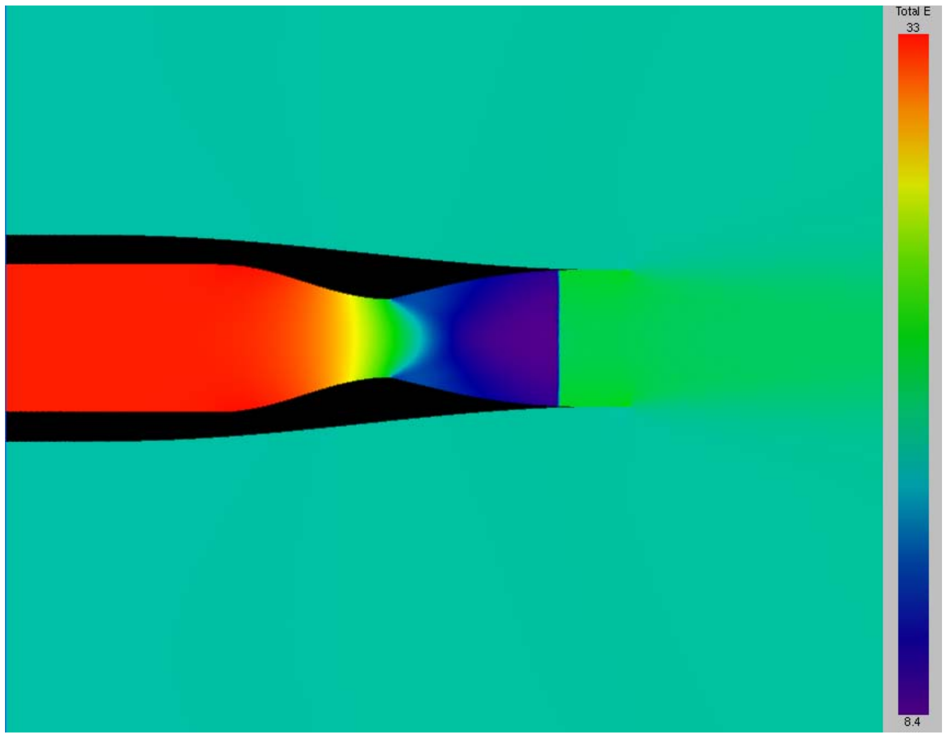


**Figure 4:6: Rocket nozzle with shock near exit**

The internal energy system was also tested using a series of different test cases.

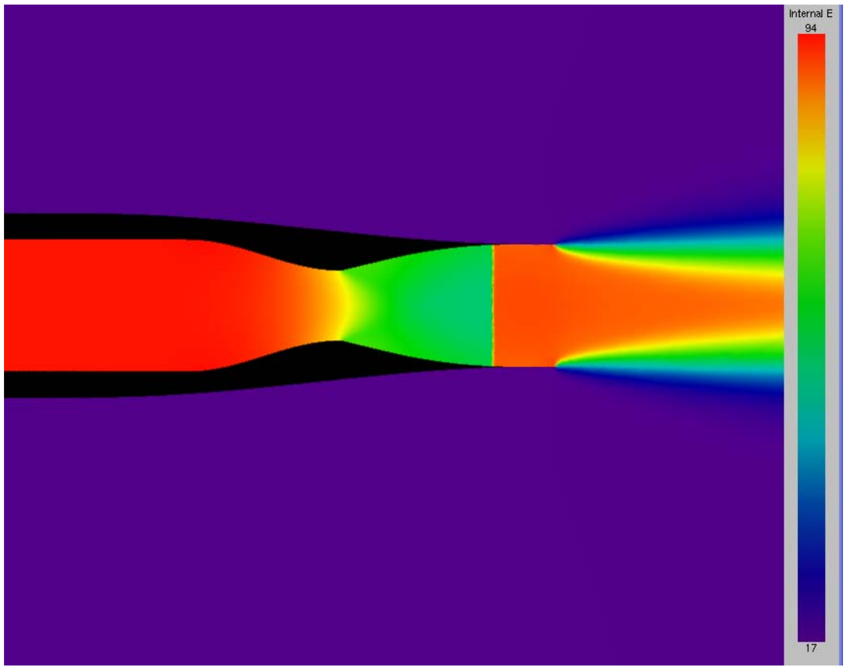


**Figure 4:7: Rocket nozzle with internal energy**

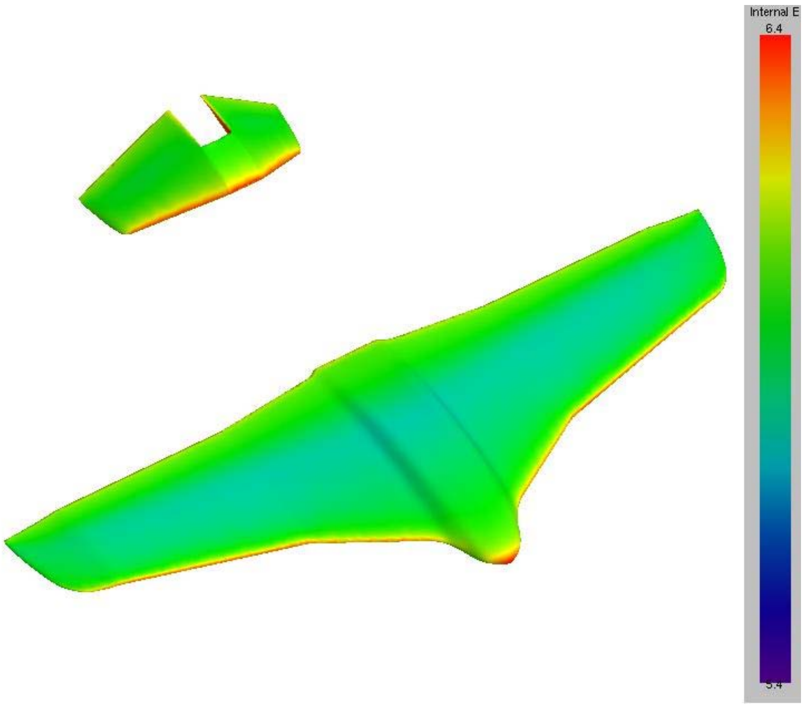


**Figure 4:8: Internal energy for NASA ARES**

It should be noted that the original internal energy display did not look like the previous figures.  When GLplot3D was upgraded, we had nothing to validate the color distribution of the object for the new variables.  All that we could check was that the minimum and maximum on the scale was indeed correct.  It was only after the creation of the VTK software, that we were able to check the color distributions.

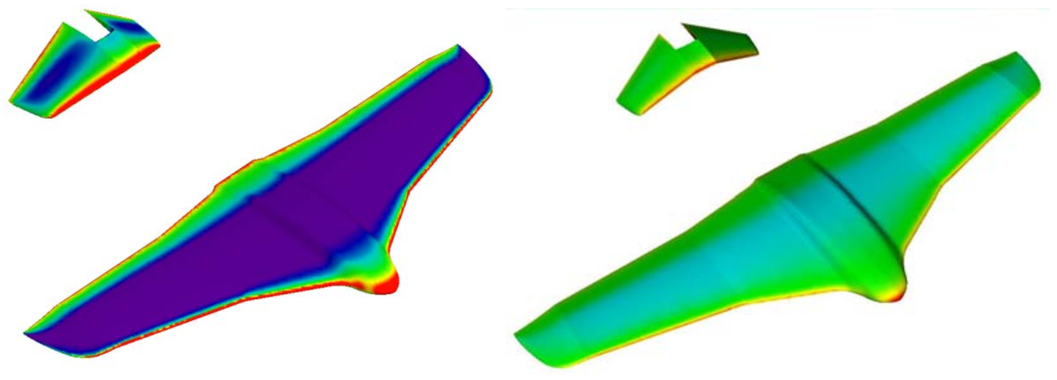


**Figure 4:9: NASA ARES internal energy comparison (GLplot3D left, VTK right)**

When comparing the VTK program with GLplot3D with internal energy, we got the same scale range, but different color plots.  The question then became which code was incorrect.  It was then found that there was an error in the internal energy code of GLplot3D.  This error was easily corrected with the help of the new VTK visualization program.

The first test case that was used in testing vorticity was a simple box geometry.  A solution file was created to allow specific velocity components to be defined.  This allowed us to know what velocity gradients were being attributed to the geometry and thus be able to calculate the vorticity in the system and validate the vorticity code.  The first case was to define a constant velocity in one direction (Vx = 1, Vy=Vz=0).

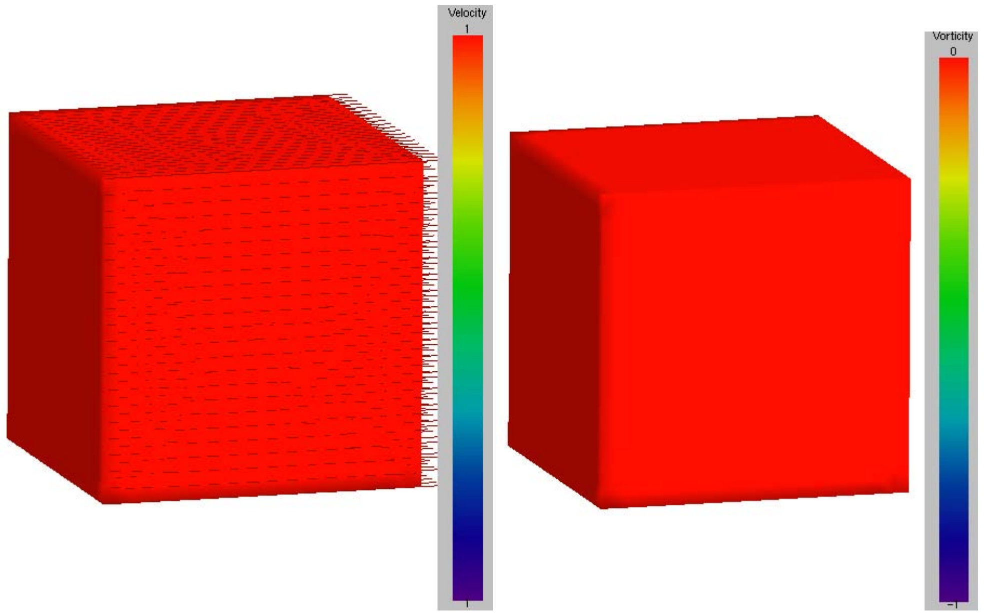


**Figure 4:10: Vx = 1, Vy = Vz = 0, Vorticity = 0**

With this velocity distribution, we should witness zero vorticity in the geometry (Figure 4:10). Another test was done by adding a velocity gradient in the direction of the velocity (dVx/dx = 0.2). This should also result in a zero vorticity value (Figure 4:11).

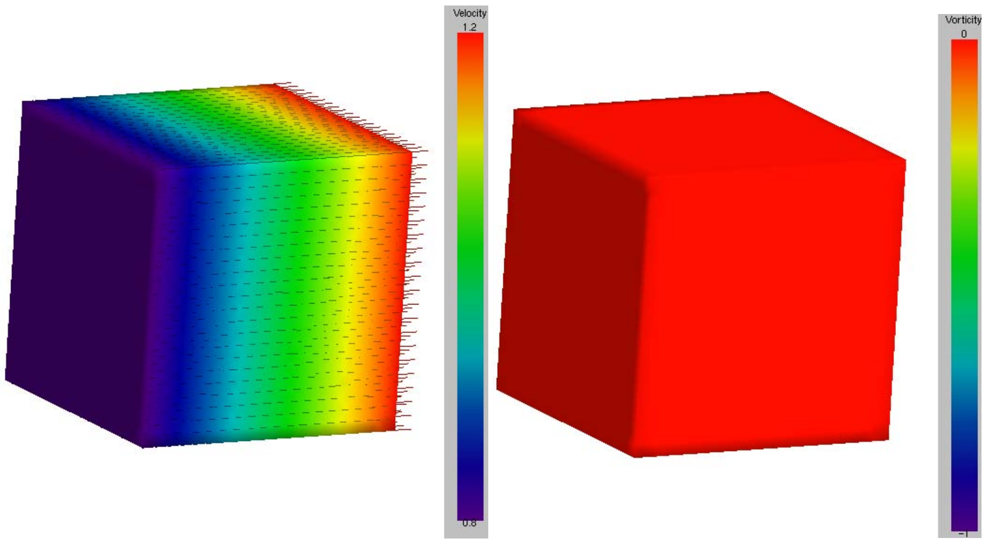


**Figure 4:11: Velocity gradient in x direction, Vorticity = 0**

A velocity gradient along the y direction of 0.2 was the next definition used in the box geometry.  This should make vorticity a constant value of 0.2 within the box (Figure 4:12).

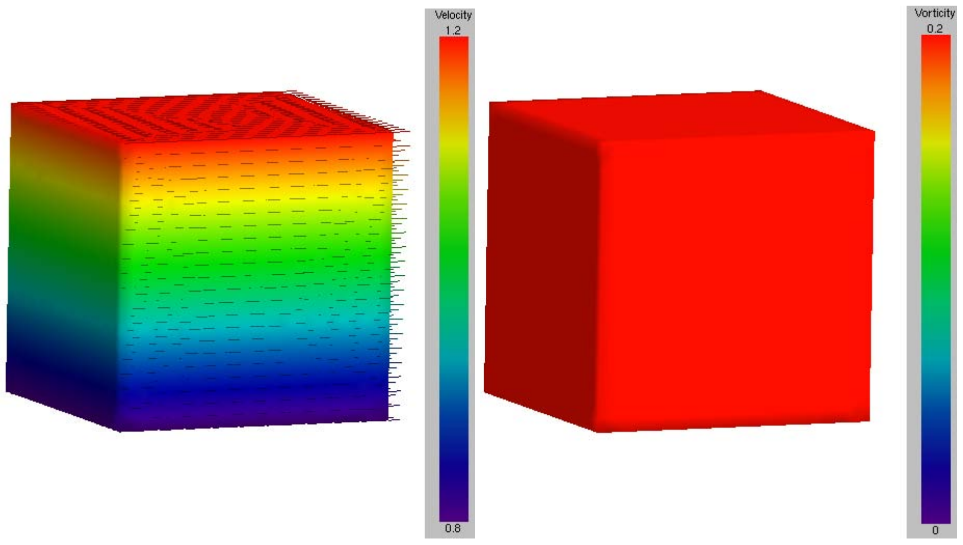


**Figure 4:12: Velocity gradient in y direction, Vorticity = 0.2**

The last validation case that was run, involved defining a nonlinear velocity gradient along the y direction, to show a linear vorticity profile in the box (Figure 4:13).

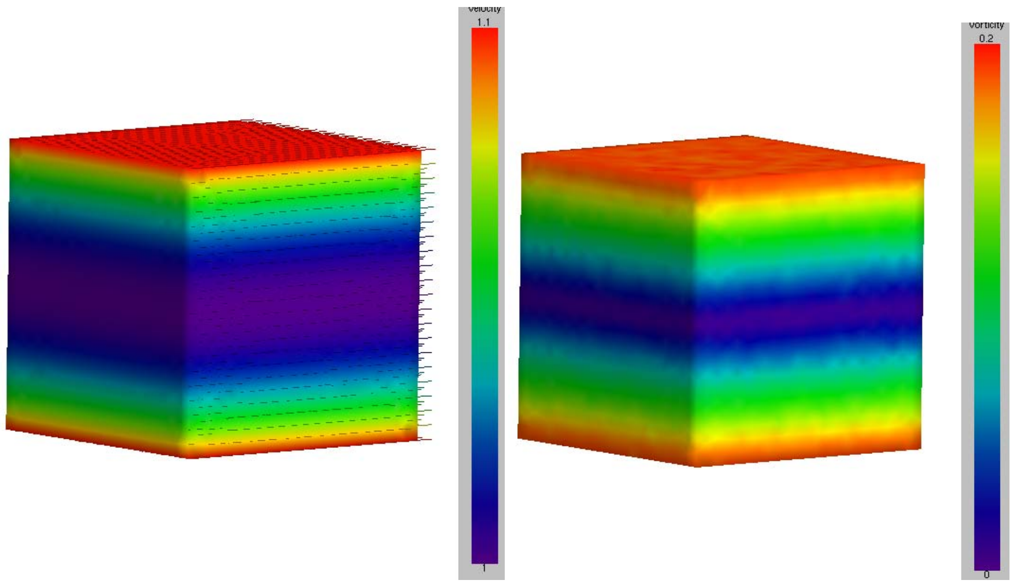


**Figure 4:13: Velocity gradient with linear vorticity**

Vorticity is currently able to be plotted in GLplot3D. However, the use of the minimum and maximum feature is needed to visualize the vorticity data. This is due to the vorticity calculation that was added to GLplot3D. Vorticity is based on the curl of the velocity and the volume of the element. In some case, GLplot3D will solve an element that has a velocity gradient from zero to a high value, which is over a very small volume of the mesh. This mainly occurs near the leading edges of the geometry for example. There is such a huge change in the velocity gradient, that the vorticity value is dominated by this single point. Thus to be able to see the normal vorticity in the flow, we need to decrease the maximum level of the vorticity using the minimum and maximum feature.

The following is a simple cavity test case where the vorticity maximum was scaled down from 96 to 0.9. Without the scale change, we were able to notice the beginning of the circulation that was created at the start of the cavity. But only after changing the scale could we see the entire vorticity in the cavity.

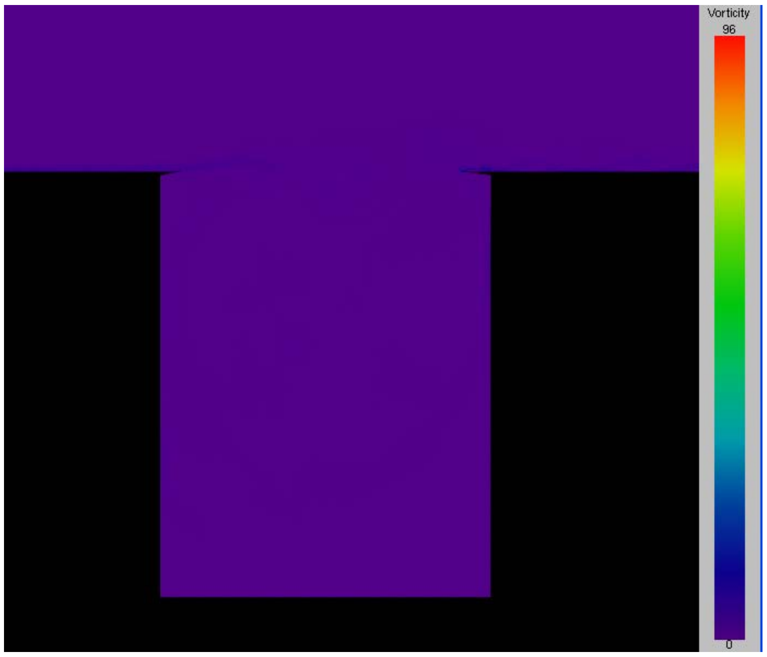


**Figure 4:14: Cavity vorticity with no scale adjustment**

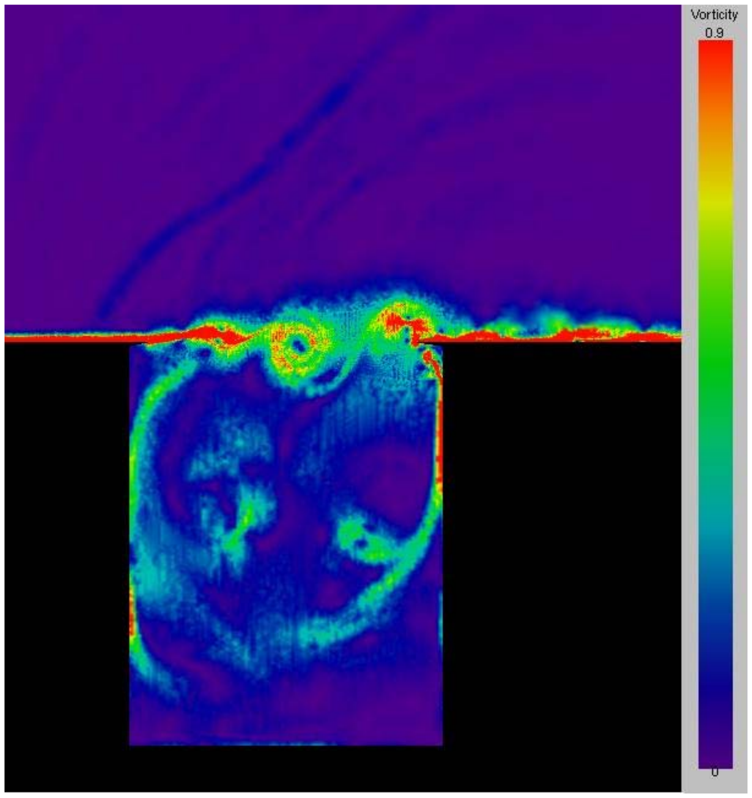


**Figure 4:15: Cavity vorticity scaled down**

The effect of the dominance of a small region can be seen in Figure 4:16. This test case is from Wagner airfoil test case. The highest velocity gradient occurs at the leading and trailing edge which dominates the vorticity scale.

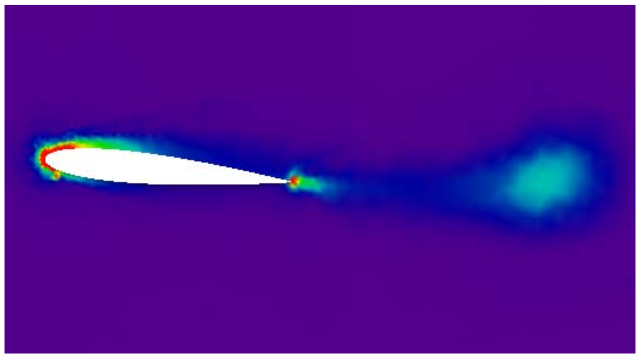


**Figure 4:16: Wagner airfoil problem showing vorticity**

Another airfoil was also used to plot vorticity. An SD 7083 airfoil was tested at a 16 degree angle of attack and plotted with the vorticity.

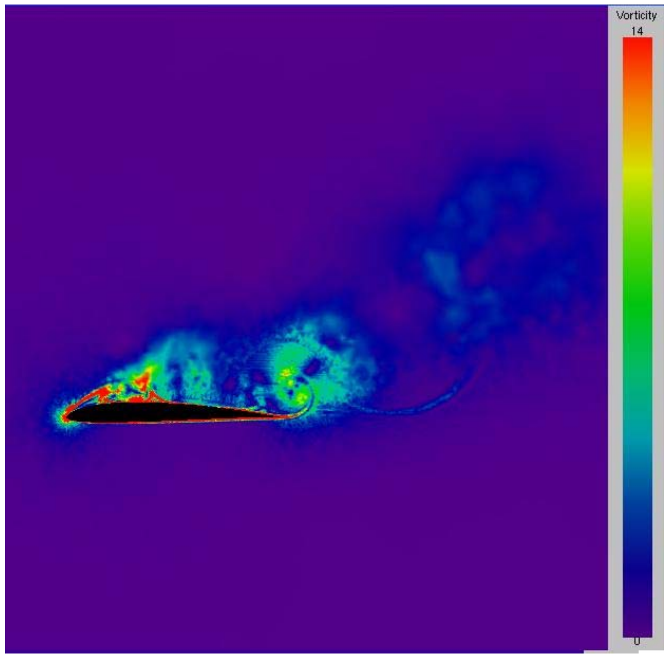


**Figure 4:17: SD 7083 vorticity**

## VTK Visualization Results

The first results we want to show from using the new VTK program is a series of geometry meshes. The first example is from a test case that was composed of two wings attached to a solid wall. This geometry contained over 4,000 nodes, and 20,000 elements.

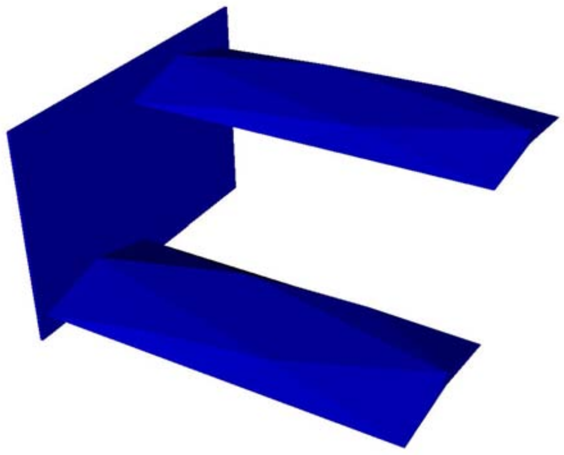


**Figure 4:18: Double Wing Geometry using VTK program**

The next case is a simple slender, finned body (missile). The missile contained over 854,000 nodes while possessing nearly 150,000 elements. The figure shows the far field with a wire mesh along with the solid missile body in the center.

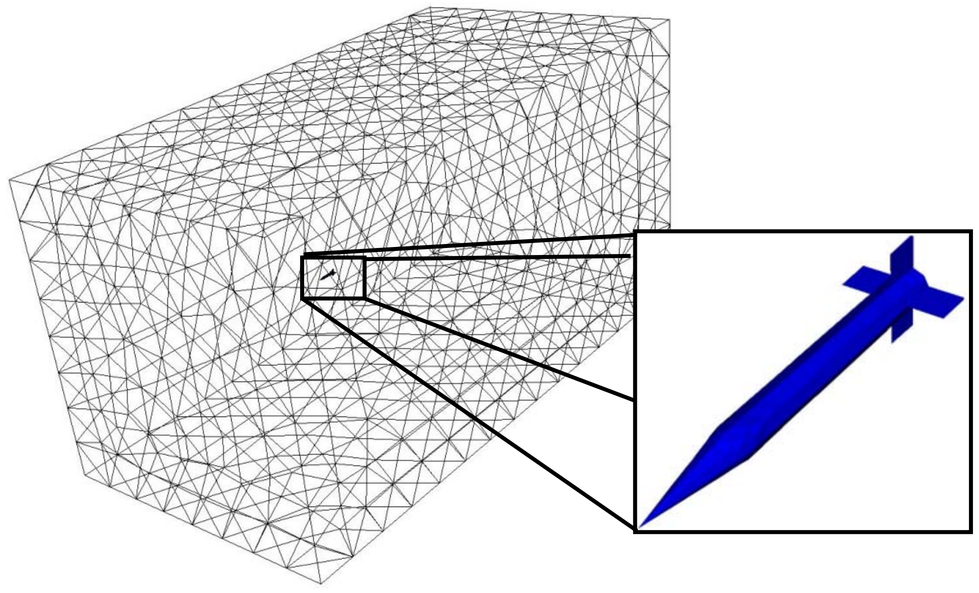


**Figure 4:19: Missile geometry showing wireframe and solid surfaces**

The final geometry is a model of the F-22 Raptor. The F-22 contained 462,000 nodes and over 2,500,000 elements.

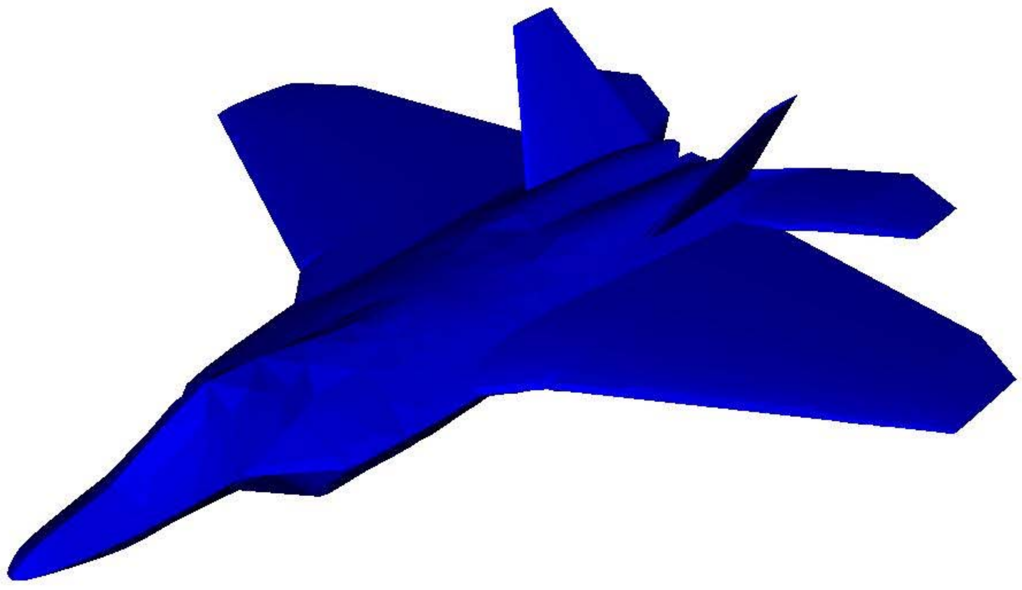


**Figure 4:20: F-22 Raptor solid surfaces**

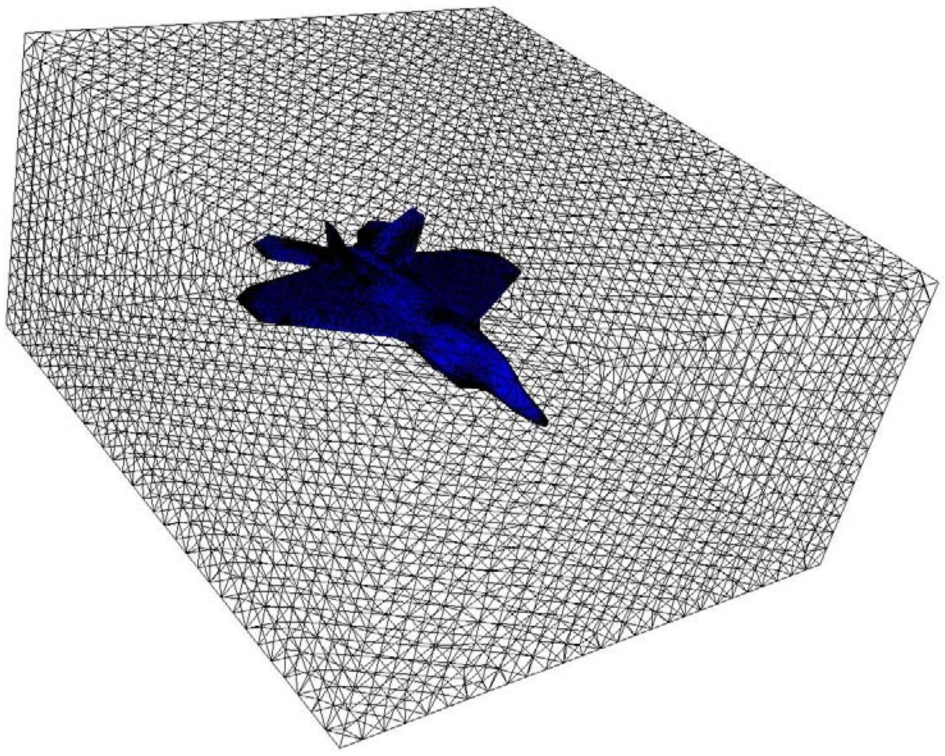


**Figure 4:21: F-22 Raptor showing solid surface and wireframe far field**

A noticeable feature in the VTK program is the texture of the mesh. The imperfections of the mesh are very noticeable. The current texture properties are using the default settings, but this feature can actually be useful when initially analyzing the test case. An example of this was on a finely meshed cone that was to be used for hypersonic test cases. When running this test case, extra shocks were being created at the tip of the cone which should not have occurred. The cone showed no imperfections when looking with a wide field using GLplot3D. The same was seen when visualizing the same test case in the VTK program.

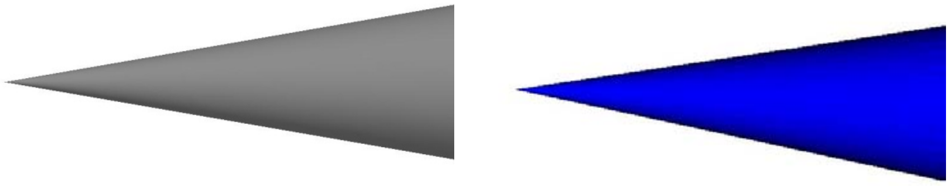


**Figure 4:22: 3D Cone (GLplot3D left, VTK right)**

However, when taking a closer look at the tip, we begin to notice some imperfections. These imperfections are more noticeable using the VTK program than with GLplot3D at a similar zoom level.

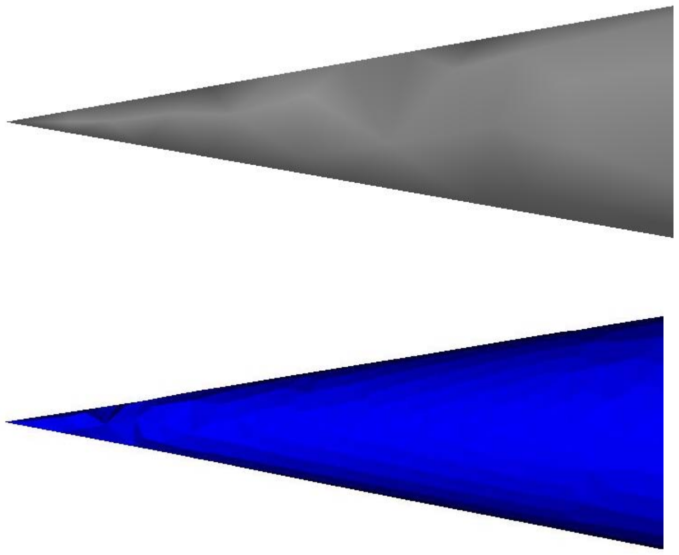


**Figure 4:23: Close up of the tip (GLplot3D top, VTK bottom)**

Only when zoomed in to a high level, and rotated can we notice the reasons for errors in the solutions.

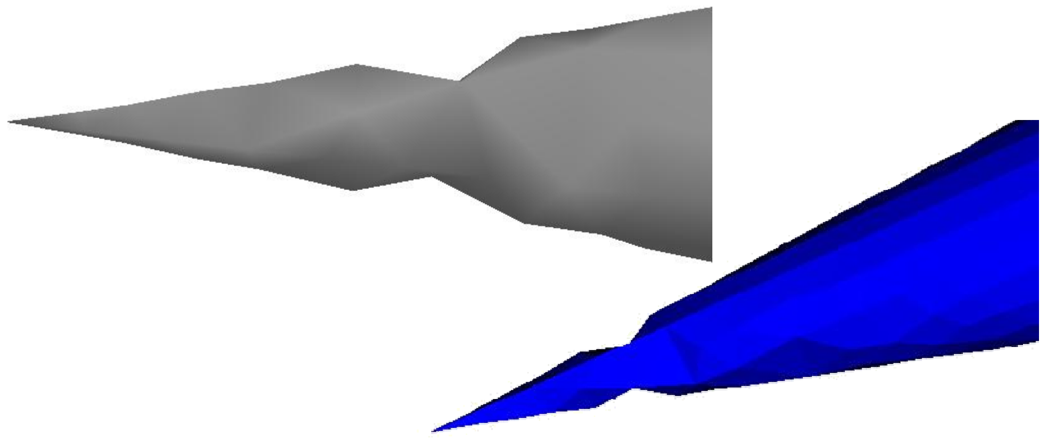


**Figure 4:24: Rotation of the cone's tip**

The next set of figures will show the results found while plotting solution files onto the geometry. One of the first test cases performed during this research endeavor was the flow visualization of a simple shock. The shock case was also used in the VTK program. The coefficient of pressure was displayed with the VTK scale is from 0.390 to -0.0314 (left) while GLplot3D is 0.39 to -0.031 (right).

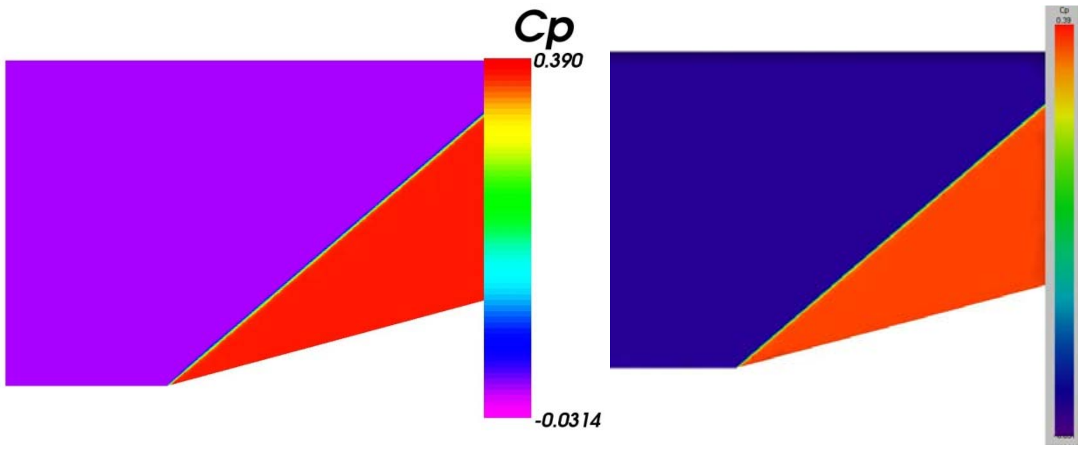


**Figure 4:25: Shock test case displaying coefficient of pressure**

This case was also validated numerically. A point was analyzed downstream of the shock to compare the theoretical values (John, 1969) with that of GLplot3D and the new VTK visualization program based on the color distribution of the displayed data.

| Property | Theory | GLplot3D | VTK |
|---|---|---|---|
| Density | 1.76 | 1.8 | 1.80 |
| Cp | 0.37 | 0.39 | 0.390 |
| \|V\| | 0.84 | 0.844 | 0.847 |
| Mach Number | 1.67 | 1.70 | 1.678 |
| Total Energy | 1.43 | 1.5 | 1.46 |
| Internal Energy | 0.46 | 0.455 | 0.457 |
| Enthalpy | 1.75 | NA | 1.79 |

**Figure 4:26: Shock case validation**

The following cases will compare the results of the VTK program with GLplot3D. One such test case is the NASA ARES Mars scout, with plotted coefficient of pressure values.

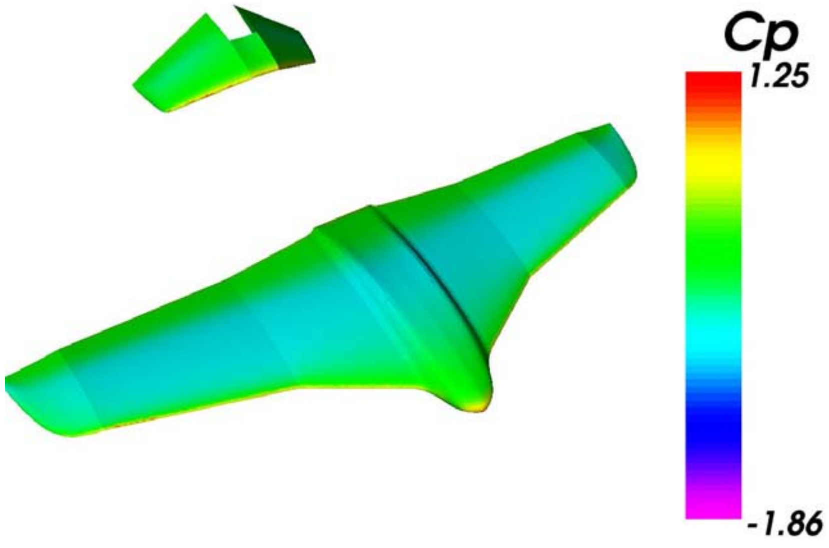


**Figure 4:27: Ares geometry with Cp solution using VTK program**

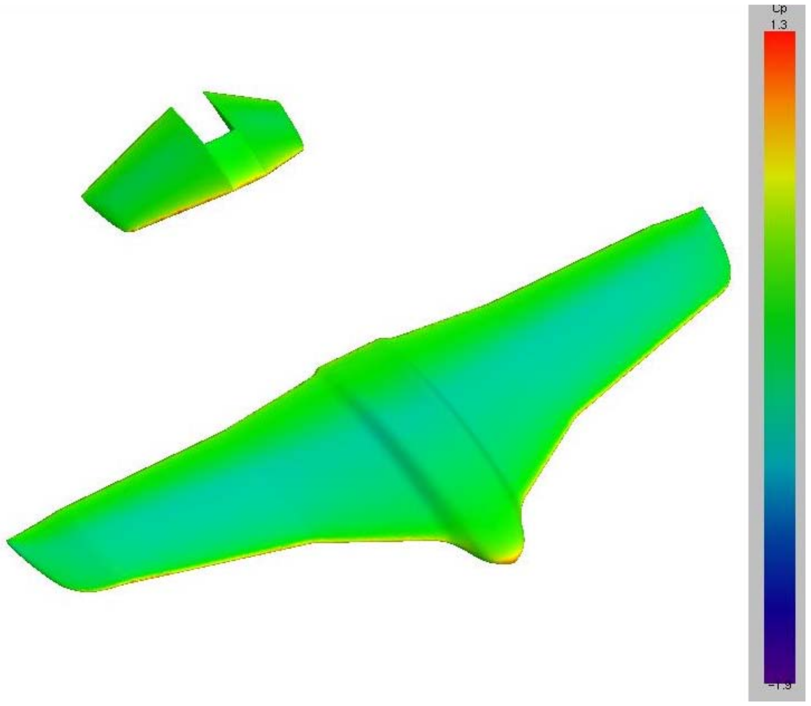


**Figure 4:28: Ares geometry with Cp solution using GLplot3D**

As can be seen there is a very slight color difference along with the minimum and maximum values on the scales.  The color is only slightly different due to the hue settings used in VTK compared to the colors scheme in GLplot3D.  By decreasing the amount of magenta (or purple) in VTK, the two figures would align more.  Also, the values on the scale used in GLplot3D round up using two significant figures, while VTK uses three significant figures.  Thus the difference between the two scales is simply due to the difference in rounding between the two codes.

As was seen in the last test case, there is a slight rounding difference between the current VTK build and GLplot3D.  But this does not cause a problem when analyzing and comparing the data.  The VTK program is plotting the correct color correlations and numerical values.  The next few test cases will further this conclusion.

The next few figures are from the NASA ARES test case.  The first comparison is looking at the plotted density values.  The VTK program shows a scale from 1.27 to 0.540 (left) while GLplot3d shows a scale between 1.3 and 0.54 (right).

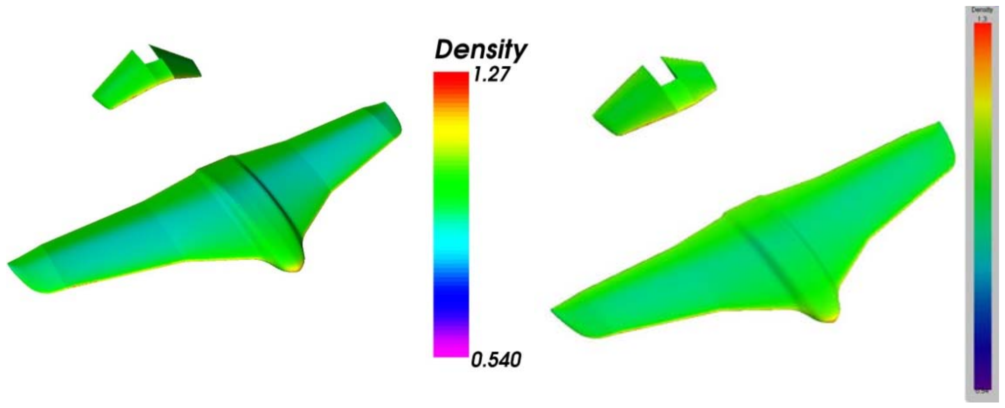


**Figure 4:29: NASA ARES density comparison**

The next test case is a modified version of the NASA ARES test case. In the previous results, the geometry has been the entire ARES, while these next few figures use only half of the ARES geometry and show a symmetry plane. The first shows a comparison of the Mach number. The VTK scale is from 0.518 to 0.00278 (left) while GLplot3D is 0.52 to 0.0028 (right).

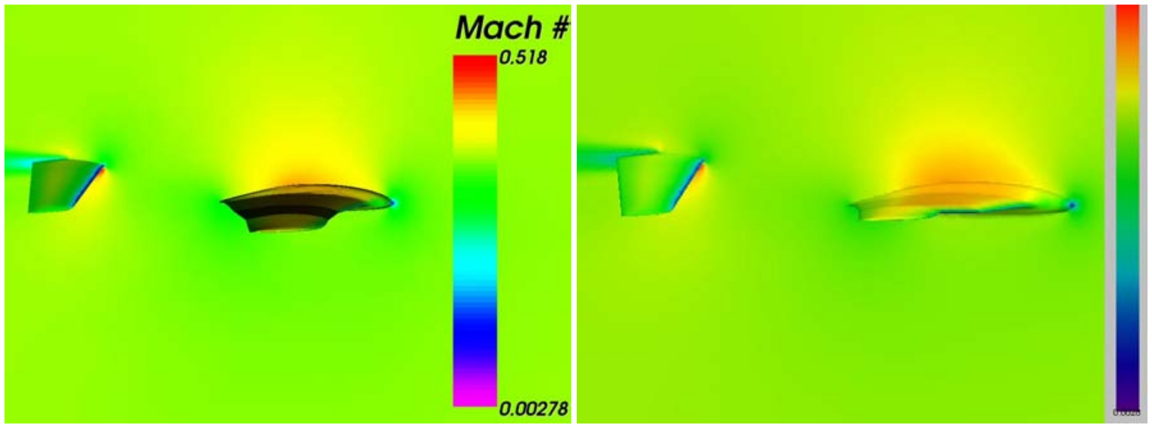


**Figure 4:30: NASA ARES symmetry plane showing Mach number**

In this test case we noticed that the VTK version shows a darker geometry. This is due to the lighting and texture effects built into VTK and left on by default. Since the wing of the ARES is coming out of the page, it shows a more shadowed effect, which can also be seen on the tail. This shadow effect can only been seen slightly in the GLplot3D version.

The next figure shows a comparison of the internal energy on the half mesh ARES. The VTK scale is from 11.4 to 11.0 (left) while GLplot3D is 12 to 11 (right).

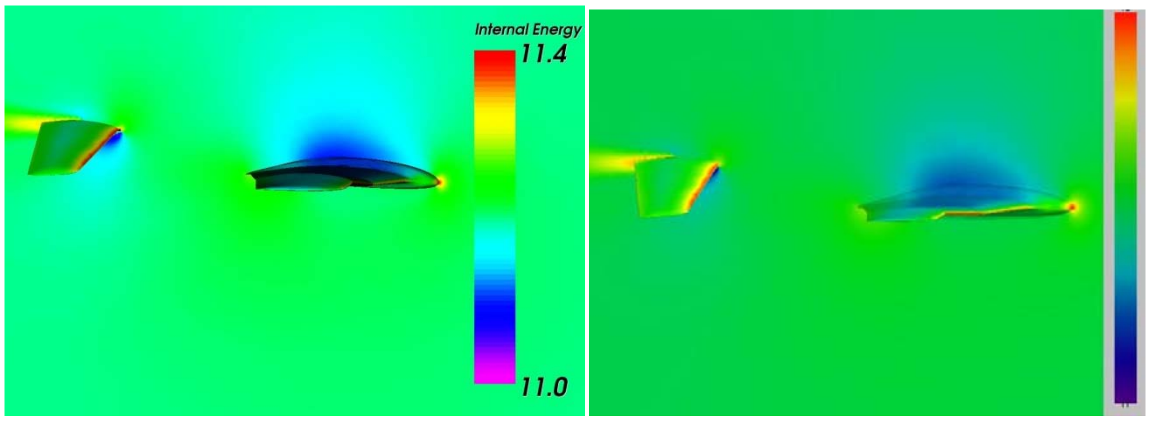


**Figure 4:31: NASA ARES symmetry plane showing internal energy**

Again, we notice the shading effects in the VTK version. However, we can still make out the color profile, and when rotated, the profile on the wing or tail becomes very noticeable.

The next figure displays the VTK program’s ability to show the wire mesh of the system. The coefficient of pressure was plotted for the ARES with the wire mesh option enabled. The VTK scale is from 0.703 to -1.37 (left) while GLplot3D is 0.7 to -1.4 (right).

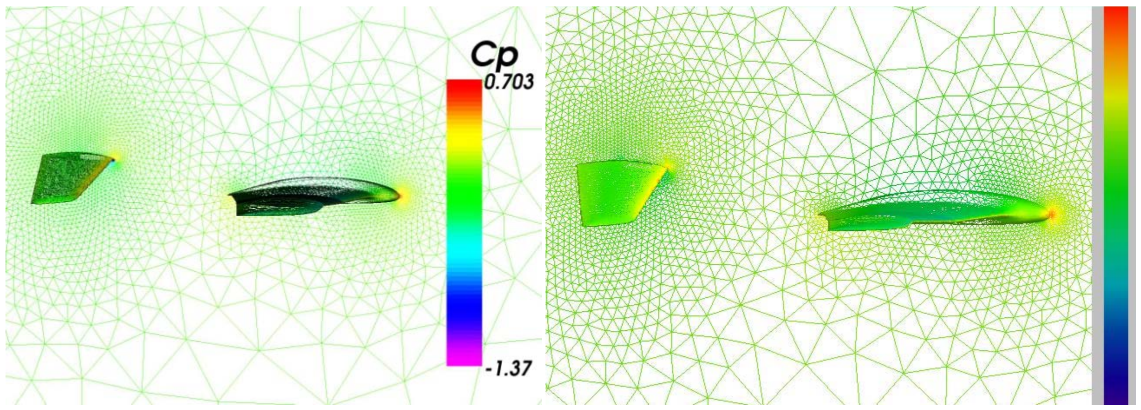


**Figure 4:32: ARES with symmetry plane utilizing wire mesh display option**

Current research is being done in the CASE Lab involving engine performance. Some simple rocket and afterburner test cases were created for use in GLplot3D and then used in VTK for validation. The next figure is from a simple rocket with a converging-diverging nozzle with an oblique shock at the exit. The coefficient of pressure was plotted with a VTK scale from 22.3 to -9.33 (left) while GLplot3D is 22 to -9.3 (right).

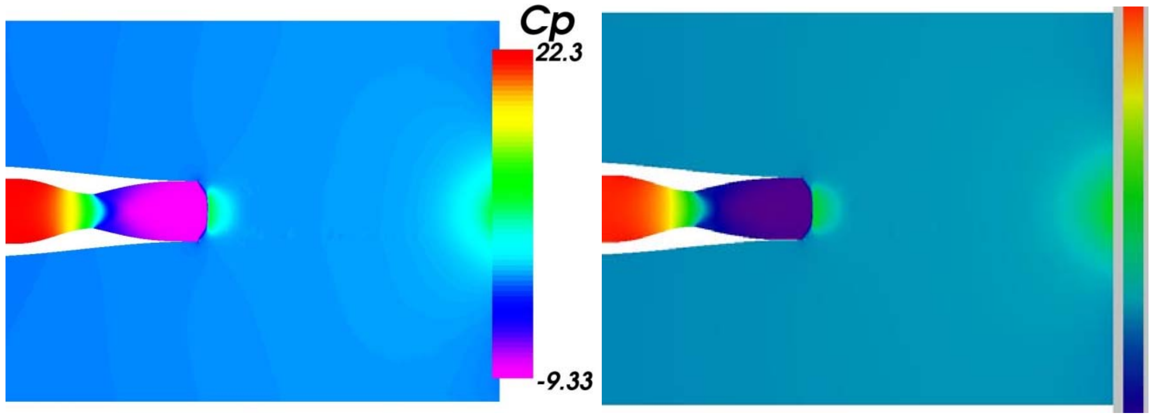


**Figure 4:33: Rocket nozzle with oblique shock showing coefficient of pressure**

An interesting occurrence can be easily seen in the VTK image with the pressure ripples in the free stream. Some of these effects can only slightly seen in the GLplot3D if we look close enough and know what to look for. The color effects in the VTK functions allow for a more noticeable pressure change.

The internal energy was also plotted for the rocket test case. With this variable, we can see the exhaust plume of the system. The VTK scale is from 93.6 to 16.9 (left) while GLplot3D is 94 to 17 (right).

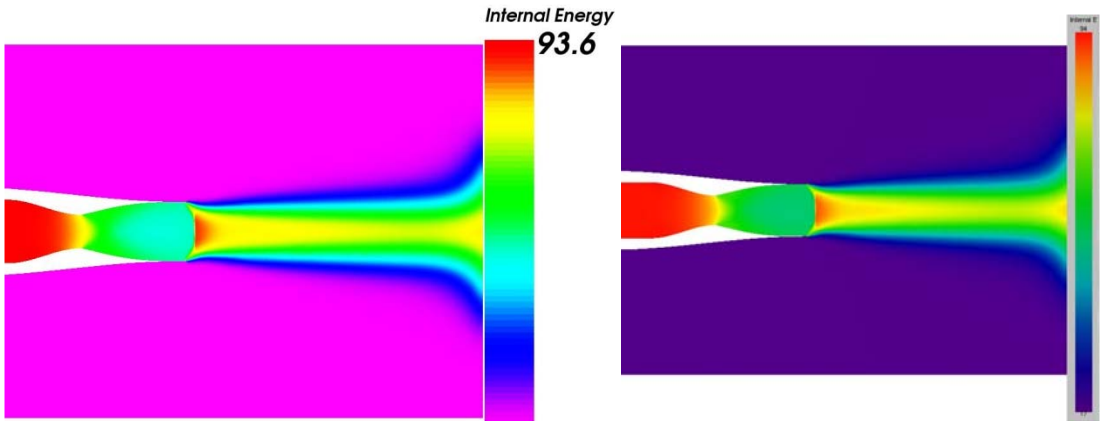


**Figure 4:34: Rocket internal energy with oblique shock at exit**

The rocket test case was also solved for the case of a shock at the exit. The VTK scale for the Mach number is from 2.07 to 0.217 (left) while GLplot3D is 2.1 to 0.22 (right).

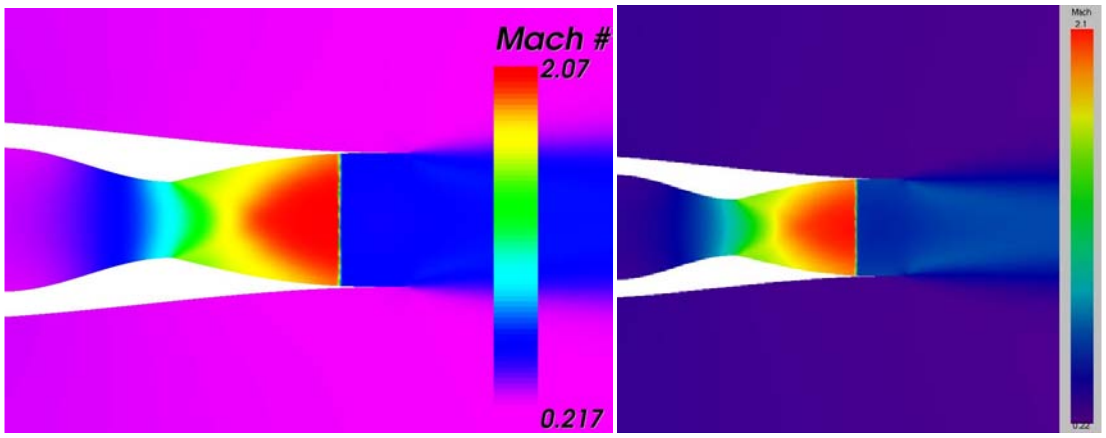


**Figure 4:35: Rocket Mach number with shock near exit**

Another field that the CASE Lab has been experimenting with is hypersonic test cases. A hypersonic vehicle test case was created and analyzed.  The VTK scale for the Mach number for this test case is from 11.4 to 6.97 (left) while GLplot3D is 11 to 6.7 (right).

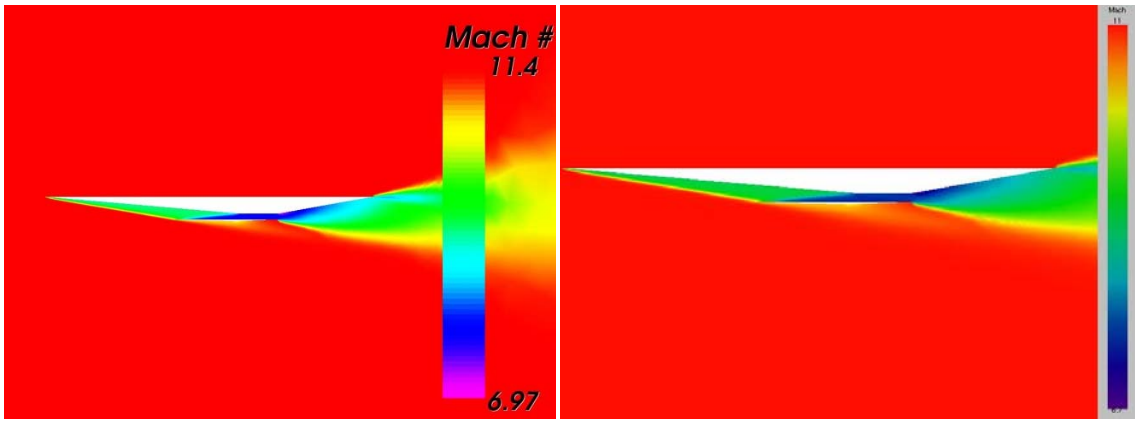


Figure 4:36: Hypersonic test case with plotted Mach number

Some of the transitions that occur in the magenta and blue color range are easier to notice in the VTK program than in GLplot3D. The same hypersonic vehicle was then plotted with the coefficient of pressure. This is another example of the how the coloring effects in the VTK code add to the noticeable effects in the test case data. The VTK scale is from 0.0925 to -0.00221 (left) while GLplot3D is 0.092 to -0.0022 (right).

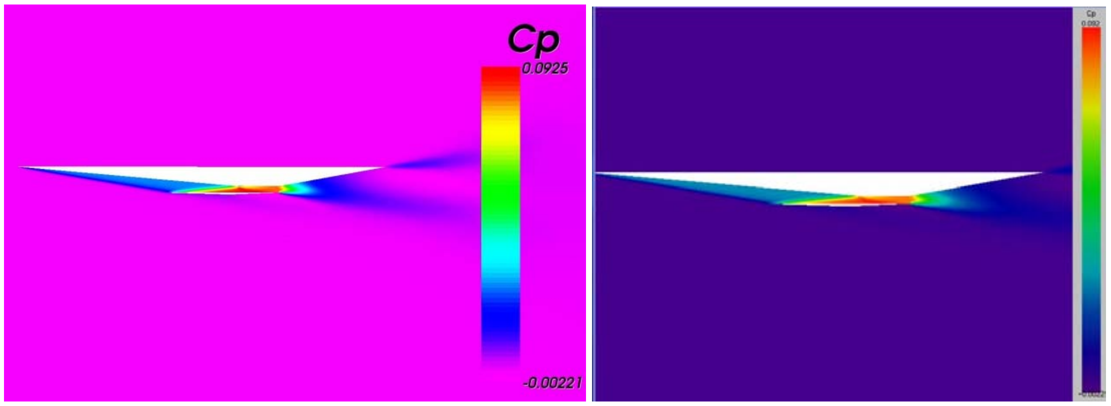


Figure 4:37: Hypersonic vehicle coefficient of pressure

The magenta color allows for an easier distinction between expansion pressures located at the lower ends of the scale, as compared to the darker purple and navy blue in GLplot3D.

A NASA supercritical 10% thick airfoil test case was added to the VTK program. This test case shows a transonic shock across the airfoil. Again, the magenta color allows for more distinction in pressure variation over the surface of the airfoil, especially near the leading edge. The VTK scale is from 1.12 to -1.81 (left) while GLplot3D is 1.1 to -1.8 (right).

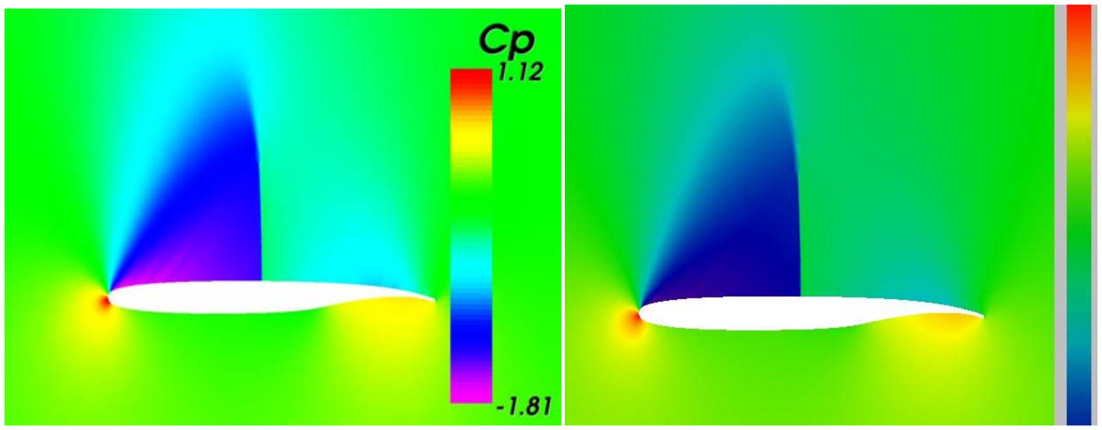


**Figure 4:38: NASA supercritical airfoil coefficient of pressure**

An SD 7083 airfoil was tested at a high angle of attack (16 degrees). The entropy distribution is more evident in the VTK version than in the GLplot3D. The VTK scale is from 0.243 to -0.0138 (left) while GLplot3D is 0.24 to -0.014 (right).

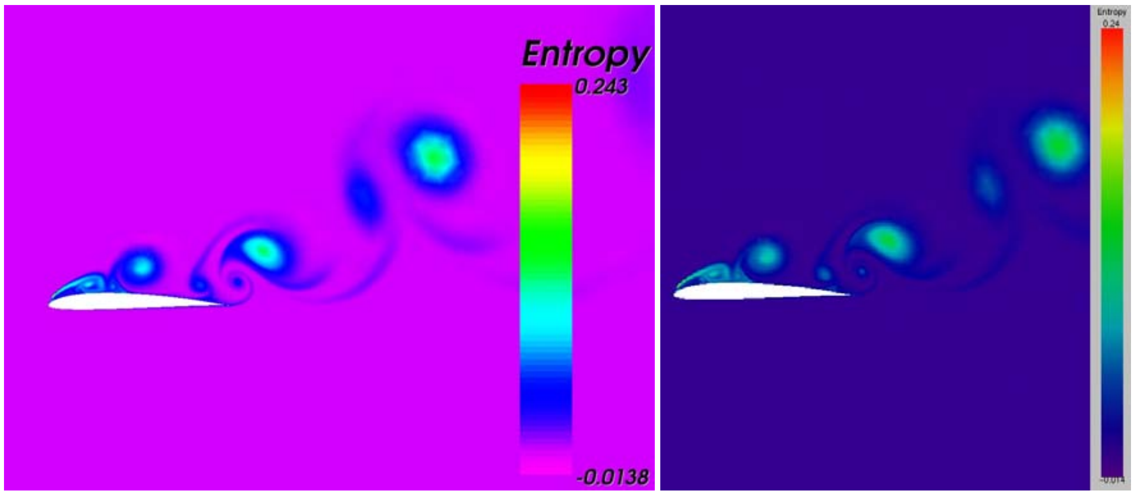


**Figure 4:39: SD 7083 16deg angle of attack entropy**

The next case is a simple cavity test case. The entropy shows the amount of circulation that is occurring within the cavity. The VTK scale is from 0.190 to -0.0174 (left) while GLplot3D is 0.19 to -0.017 (right).

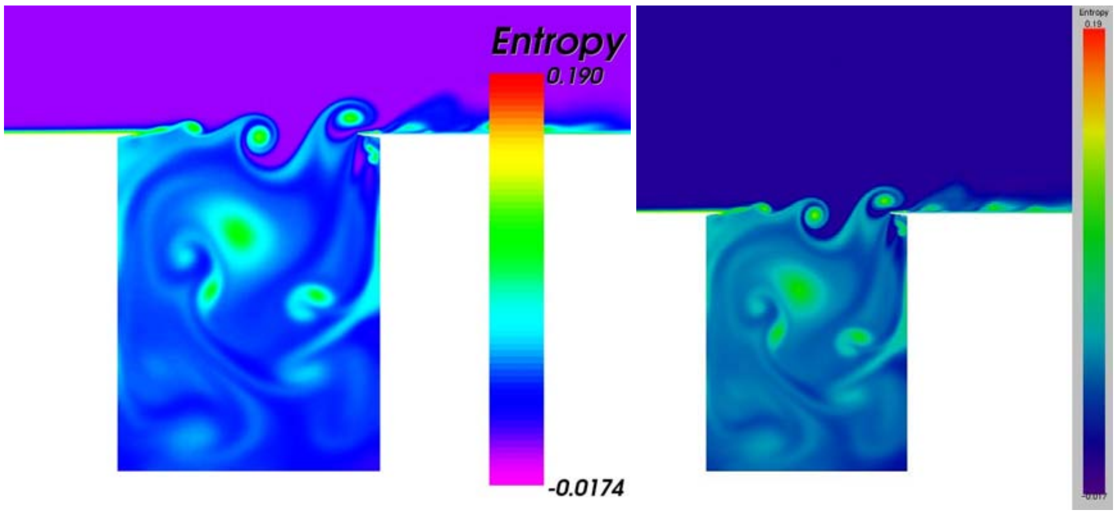


**Figure 4:40: Cavity test case with entropy**

An F18 test case was also plotted in VTK and GLplot3D for comparison. The coefficient of pressure was plotted for the test case. The VTK scale is from 1.55 to -0.835 (Figure 4:41) while GLplot3D is 1.5 to -0.83 (Figure 4:42).

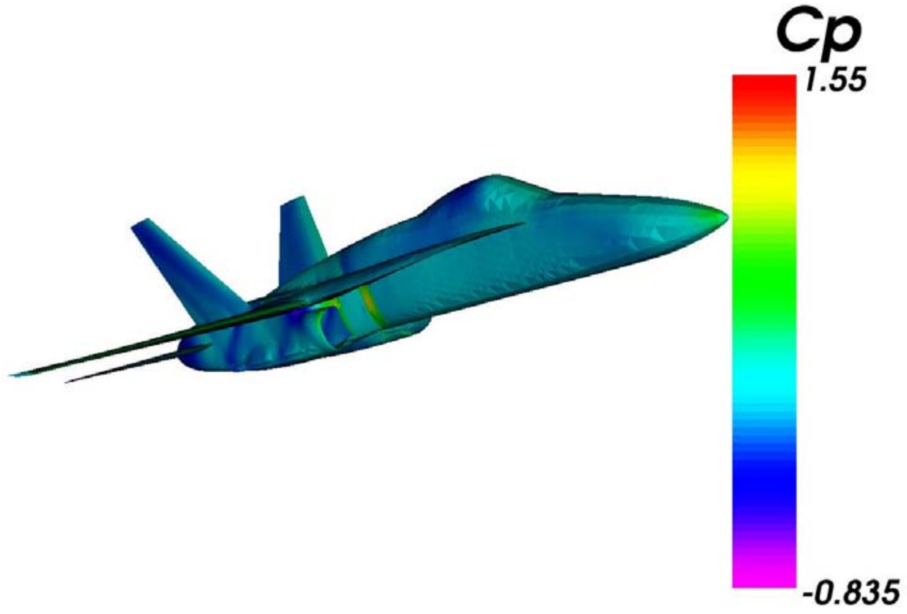


**Figure 4:41: F18 coefficient of pressure using VTK program**

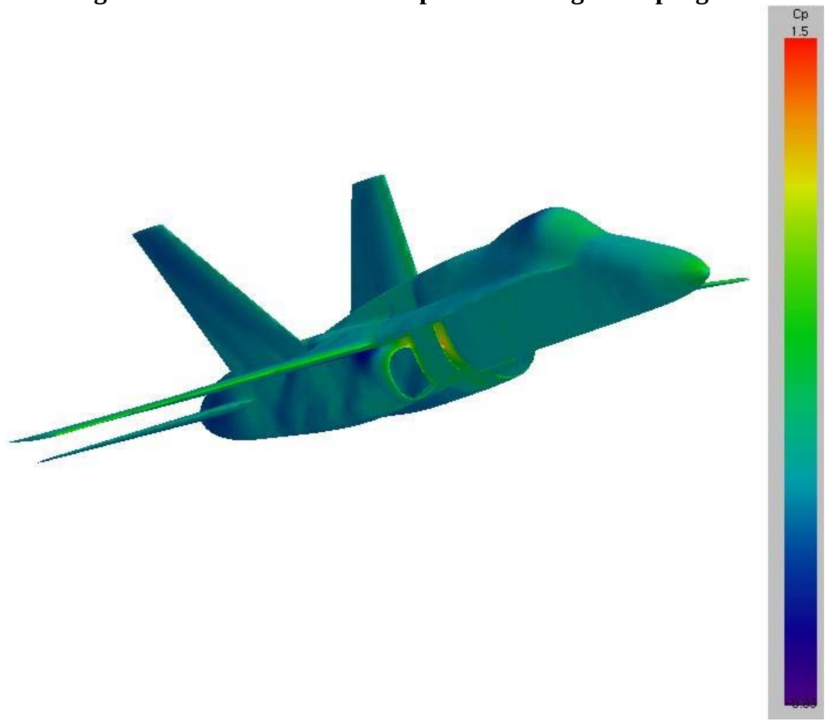


**Figure 4:42: F18 coefficient of pressure using GLplot3D**

From this same test case, we wanted to show the another ability that can be compared with GLplot3D: the ability to plot the nodes.  GLplot3D has the option to show specific surfaces and nodes.  The nodes were left on for the F18 case to show the comparison when viewing the data coefficient of pressure data on the geometry.  Figure 4:43 is the same test case as in Figure 4:41 and Figure 4:42 that were previously shown.

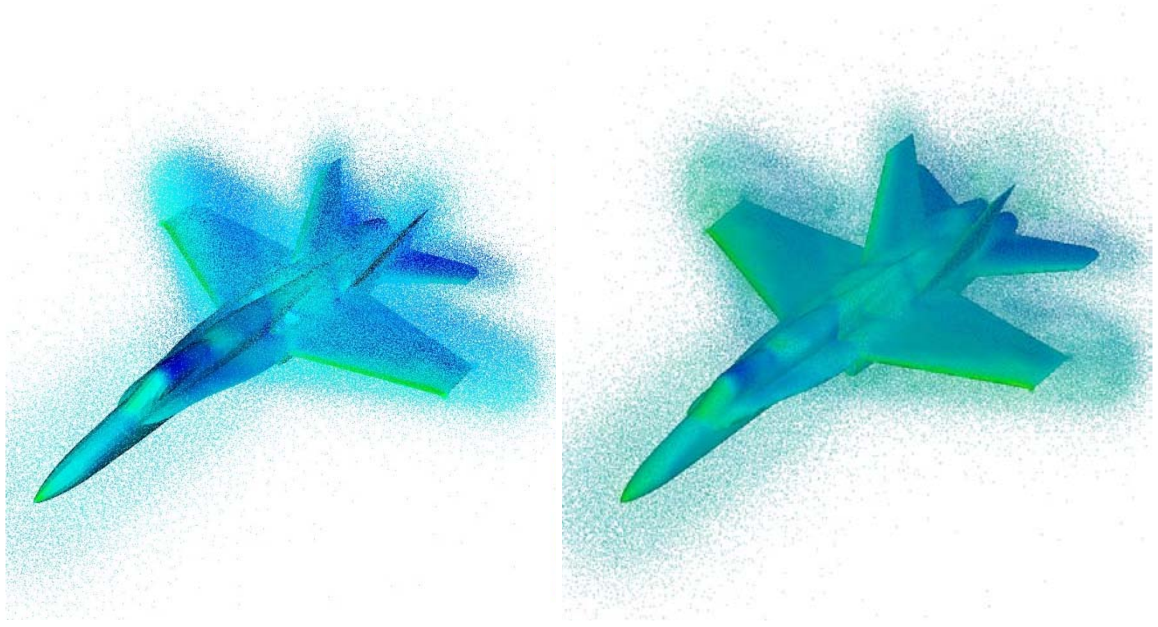


**Figure 4:43: F18 coefficient of pressure showing solid surface and nodes**

# CHAPTER V

# CONCLUSIONS

In this chapter, we will further discuss the results and how they pertain to the objectives set out in the beginning of this thesis. As well as make recommendations for the further progress of visualization in the CASE Lab.

## Discussion of Results

At the beginning of this thesis, four objectives were stated that were the focus of this thesis. Here we will discuss the outcomes of these objectives and show that they were completed.

- Successfully created new variables and a new feature for GLplot3D
    - Added velocity magnitude, total energy, internal energy, and vorticity
    - Added minimum/maximum feature to plots and scales
- Researched the necessary background in order to begin programming using the VTK architecture
    - Demonstrated basic hardcoded cases
    - Demonstrated reading in generic geometry
    - Demonstrated read and plotting solution properties
    - Expanded plot variables to compete with many of the current variables in GLplot3D
- Successfully created the first phase towards a standalone visualization program
- Created a list of recommendations for the continuation of the VTK visualization software

The first objective was to update GLplot3D with new variables and new features. GLplot3D had four new variables successfully added to its capabilities (velocity magnitude, total energy, internal energy, and vorticity) along with the ability for a user specified minimum and maximum option for the outputted data.

The second objective was to learn the basics of the VTK programming methodology. In order to learn the basics of VTK, we needed to not only read sources, but also be able to gain hands on experience with the VTK functions. By working with examples and making minor changes to the code to witness the effects, we were able to learn how to program using VTK. This later evolved to the ability to add specific code structures that would be used CASE Lab test cases.

The third objective was to create a basic visualization program using VTK to show comparable capabilities to that of GLplot3D. This was shown to be successful in Chapter 4 were we discussed the current abilities of the new VTK visualization software. The code is not meant to be a full replacement for GLplot3D at this stage of development. For the purpose of this thesis, the VTK visualization program is to serve as a proof of concept that VTK can be programmed to have the same capabilities of GLplot3D as well as have new features.

The fourth objective was to determine specific recommendations for the continuation of the VTK visualization program. These recommendations are discussed in detail in the next section.

The overall goal of this thesis was to provide the foundation for continued work in CFD visualization. This foundation has now been set for a future dissertation. The work completed with GLplot3D allowed for the understanding of incorporating finite element solutions into visualization. This understanding also helped in the ongoing development for new visualization software using VTK. The current framework of the VTK program, has given us a starting point for creating a better visualization program to incorporate the new research projects found in the CASE Lab, including the future ability to remesh geometries during a solution run.

## Recommendations

In this section, we will discuss some of the recommendations for the current two visualization programs: GLplot3D and the VTK program. These recommendations are based on the current work that has been performed on these two programs and the work that will be needed to achieve the vision that was originally set for this project.

### GLplot3D

GLplot3D has been updated to handle the current visualization workload of the CASE Lab, but use of GLplot3D should slowly phased out as more features are brought online with the VTK program. The process of phasing out GLplot3D will not be a short process, as the current VTK software is not ready for distribution. For GLplot3D, the recommendation is to continue to use it in conjunction with the VTK software to help in continual development of the VTK counterpart. As new features and further development is made with the VTK software, GLplot3D will be used as validation of the some of the newly added capabilities. After the VTK software is built to satisfy the purposes CASE Lab and other engineers (such as NASA Dryden and Advanced Engineering Solutions, AES), the VTK package will be distributed to users and will transition to the primary CASE Lab visualization tool.

## VTK

Variable Array Sizes: For the vision with VTK to be reached, the software needs to be expanded and modified to replace GLplot3D and beyond. One of the first recommendations is to implement variable arrays that can be resized within the code and not have to be hard coded based on the test case. These variable arrays need to be created and then resized based on some of the read dimensions in the geometry file; such as resizing the array to correspond to the number of nodes in the test case. There are currently two ways that are seen to approach this problem. The first is to create a resizing subroutine into the code itself. This will use only C++ coding structures and functions to resize the array based on the read data. However, newer C++ compilers may already have a resizing function for arrays integrated into the compiler. Due to the age and restrictions of GLplot3D, such compilers have not been reinstalled to test this theory, and will thus need to be looked into when continuing the progress of the VTK software. The second option is to use one of the VTK array members. Some of the VTK array members already have a resize function that can be called by the program. This will help with the coding, and may also allow for fewer array classes to be declared and used since the data arrays are already in a VTK array format to be used by the actors and mappers.

VTK User Interface: The VTK software will also need to incorporate a user interface. This will help with changing between different variables, and turning on or off different features which will be discussed further on. The user interface will need some basic features. These include a construct to allow the user to determine the test case that is to be loaded, construct to determine which variable is to be plotted along with the types of geometry to display (nodes, surfaces, segments, boundary wall, etc).

Expand File Types: The implementation of new file types will also need to be added into the VTK software. The vision of the VTK visualization software is to be capable of reading not only all the file types currently used by the CASE Lab but to be able to add new file types later as they are infused into the CASE Lab system. Currently such file types include .vec files for reading in elastics, other geometry files (.g3d and .cfs) and their corresponding solution files (.unk, .un#, .unk.#, etc).

Unsteady Time Slices: Currently, the VTK program can read in a steady solution file (.unk). The ability to read in multiple solution files (.un#) will need to be added to allow for unsteady solutions to be displayed. The program will need to be capable of reading in the entire set of data from the .un# files and display the solution sets one at a time. The program will need to display one solution and then progress to the next solution file at the request of the user via either a keyboard command or user interface.

New Variables: Other variables will need to be added for display on the geometry. Velocity vectors and vorticity are two variables that are currently able to be displayed by GLplot3D and will need to be created in the VTK software. Velocity magnitude has already been shown to work, however vectors will need to be created in the program using a VTK sanctioned routine and added to the nodes. Other quantities, such as wall shear friction, wall heat flux, and turbulence quantities can be added as these properties are added to the CFD packages.

New Features (Min/Max): There are also other features that need to be implemented to allow the VTK software to meet and surpass the current GLplot3D. First, a minimum and maximum feature should be added similar to the one that has been added to GLplot3D. This needs to be added with the user interface, to allow the user the ability to determine the minimum and maximum data value to be displayed. This should be similar to GLplot3D, where any value larger than the maximum will display red, while any value lower than the prescribed minimum be purple, and then be able to convert back to the original values.

New Features: Cut Planes and Line Graphs: A second feature is the addition of cut planes. As with the features in GLplot3D, we want the ability to select a single plane and be able to visualize the values just on this single plane. VTK has many different cutting techniques that can be used to a cross-section of data and plot the data set on the plane. Another feature is the ability to plot values on a line graph. VTK has s specific function called *vtkXYPlotActor* which allows the ability to create x-y plots from one or more sets of data. This feature would allow for the generation of graphs without having to manually input solution data into plotting programs such as Excel: property plots, load histories from .lds file, and convergence from the residuals, .rsd, file.

Streamlines: Streamlines would also be a nice addition to the CASE Lab visualization. VTK already has preprogrammed functions for such a feature. The user should have the ability to add, quickly place, and delete starting points for streamlines.. Along with streamlines, having the ability to plot particle traces should be added.

GLplot3D has a particle trace subroutine in its code, however, this function has not shown favorable results and takes large processing times on the order of days.

Snapshots: Currently GLplot3D has the ability to take a snapshot of the current view and create a file in the current working directory. However, the format of the picture file must be opened in another program and then converted to a workable format for use in presentations, papers, reports, etc. These pictures are also at a reduced quality, which does not reflect what is seen by the user on the screen. To minimize pixilation, a Window's screen capture is often used. The VTK program needs to also possess the ability to create a picture file from the current rendered view. However, if possible the snapshot file should already be in a ready to use format (jpeg, bmp, gif, etc) with the same quality seen on the screen.

Stream Tubes: Not only are we getting the VTK software to a point where it has all the functions of GLplot3D plus other features, we are also looking at current CASE Lab research and ways to begin implementing new CFD progress into the VTK visualization software. One current research topic is the addition of engine data. Engine research will allow the CASE Lab's solvers to incorporate engine performance parameters into the solution (Moffitt, 2010). New boundaries have been created to simulate inlets and exits for engines and allow for the addition of mass and energy into the flow as an actual engine. New features such as stream tubes could be added to help visualize the flow in and out of the engine. The stream tubes would be attached to the engine inlets and exits and trace up and down stream along the edge of the flow that passes though the engine.

Viscous/Turbulence Data: Current modifications are also being made for viscous and turbulence models. Viscous and turbulence variables are being added into the CASE Lab solvers and will be stored in the solution file types (.un# and .unk). The VTK program will need to be capable of reading these values and plot them correct. We recommend that when these data sets are stored in the solution files, the values be simply added to the end of the already existing data arrays. This will make integration in the visualization program smoother since the program will be reading from the same array instead of an additional file, or different array structure. Additional arrays will need to be added to the visualization program, but the collection of the data from the solution file will be easier to implement in future versions of the VTK software. The VTK program will also need to test for the presence of viscous and turbulence data, thus a flag needs to be added to solution file to signify the presence of such data.

Non-Inertial Reference Frame: The VTK software should also have inertial and non-inertial reference frame options for use with dynamic test cases. New methods for enhancing dynamic models are being researched in the CASE Lab, and the VTK software should be capable to adapt to the coming changes. By giving the option of inertial and non-inertial reference frames, we will be able to visualize an object motion during a dynamic maneuver and not simply plot values on a graph, and plot velocities and other properties in their respective reference frames.

Multiple Meshes: Along with this addition for dynamic test cases, we in the CASE Lab are researching the ability to read in multiple geometry files for a single test case. That is, to be able to create a “remeshing” program within the CASE Lab solvers and visualization software codes.  This remeshing system will read in specified geometry movements and incrementally move through the motion while creating a new mesh with the system to visualize the effects of moving geometries.  Such geometries include elevator, aileron, or rudder deflections, landing gear doors, cavity changes, flapping wings, and more.  At current, the thought is to change the geometry file through the motion, thus the visualization software will need to be able to read and store multiple geometry files.

These recommendations will allow for a quicker starting phase for the completion of the VTK visualization program and future dissertation.

# REFERENCES

Avila, L. S. (2006). *The VTK User's Guide: Updated for VTK Version 5*. Clifton Park, NY: Kitware.

Coltrane, J.J. and Arena, A.S. (2005). CFD Modeling of NASA’s ARES Platform. 36th Annual Lunar and Planetary Science Conference.

Cowan, T. J. (2003, August). "Finite Element CFD Analysis of Super-Maneuvering and Spinning Structures." *Ph.D. Dissertation*, Oklahoma State University, Department of Mechanical & Aerospace Engineering.

Emmons, H. W. (1958). *Fundamentals of Gas Dynamics*, Boston, MA: Princeton University Press.

John, J.E.A. (1969). *Gas Dynamics*. Englewood Cliffs, NJ: Prentice Hall.

Kok, A.J.F. and Liere R. (2007). “A multimodal virtual reality interface for 3D interaction with VTK”. Springer-Verlag London Limited.

Moffitt, N. J (2010). “Galerkin CFD Solvers for use in a Multi-Disciplinary Suite for Modeling Air-Breathing Hypersonic Vehicles and Other Classes of Advanced Flight Vehicles” Doctorate Preliminary, Oklahoma State University, Department of Mechanical & Aerospace Engineering.

Maple, R.C., Osterday, R., Vickery, R. (2006). “Fluent VTK Extractor”. Washington DC. IEEE Computer Society.

O'Neill C. R. and Arena A.S. (2005, January). "Aircraft Flight Dynamics with a Non-Inertial CFD Code" *AIAA Paper 2005-0230*, AIAA 43th Aerospace Sciences Meeting and Exhibit, Reno, NV.

Papademetris, X., Joshi, A. (2009). *An Introduction to Programming for Medical Image analysis with the Visualization Toolkit*. Department of Biomedical Engineering, Yale University.

Schroeder, W.J., Martin, K., and Lorensen, B. (2006). *The Visualization Toolkit: an Object-oriented Approach to 3D Graphics*. Clifton Park, NY: Kitware.

Schroeder, W.J. (2005). “Framework for Visualizing Higher-Order Basis Functions”. Clifton Park, NY: Kitware.

Squillacote, A. H. (2007). *The Paraview Guide: A Parallel Visualization Application*. Clifton Park, NY: Kitware.

Stephens C. H. and Arena A.S. (1998, April). "Application of the Transpiration Method for Aeroservoelastic Prediction Using CFD." *AIAA Paper 98-2071*, 39th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference, Long Beach, CA.

Rew, R. (2009). “The NetCDF C Interface Guide.” University Corporation for Atmospheric Research.

"VTK: VTK 5.4.2 Documentation." *VTK - The Visualization Toolkit*. Web. 13 March 2009. <http://www.vtk.org/doc/release/5.4/html/index.html>.

APPENDICES

## APPENDIX I –CONE EXAMPLE

```
/*==================================================
================
Cody Pinkerman

Modified VTK Example to read in nnd, nel, & nsg from nc3d file
and generate a cone structure
==================================================
==============*/

// first include the required header files for the vtk classes we are using
#include "vtkConeSource.h"
#include "vtkPolyDataMapper.h"
#include "vtkRenderWindow.h"
#include "vtkCommand.h"
#include "vtkCamera.h"
#include "vtkActor.h"
#include "vtkRenderer.h"
#include <string.h>

#include "NetCDF\NetCDF.h" //to use nc commands

//Read in data from text file

int r, h, res; //variables to read in

void ncstatus(int status)
{
   if(status != NC_NOERR){
   fprintf(stderr, "%s\n", nc_strerror(status));
   exit(-1);
   }
}

void readme(char *name)
{
         int ncid;
         int nndVar, nelVar, nsgVar;

         char filename[256];
         strcpy(filename,name);
         strcat(filename, ".siz");

         ncstatus(nc_open(filename,0,&ncid));


         ncstatus(nc_inq_dimid(ncid, "nnd", &nndVar));
```

```
            ncstatus(nc_inq_dimid(ncid, "nel", &nelVar));
            ncstatus(nc_inq_dimid(ncid, "nsg", &nsgVar));

            ncstatus(nc_inq_dimlen(ncid, nndVar, (size_t *) &r));
    ncstatus(nc_inq_dimlen(ncid, nelVar, (size_t *) &h));
    ncstatus(nc_inq_dimlen(ncid, nsgVar, (size_t *) &res));

            ncstatus(nc_close(ncid));
}


// Callback for the interaction
/**
class vtkMyCallback : public vtkCommand
{
public:
  static vtkMyCallback *New()
    { return new vtkMyCallback; }
  virtual void Execute(vtkObject *caller, unsigned long, void*)
    {
      vtkRenderer *renderer = reinterpret_cast<vtkRenderer*>(caller);
      cout << renderer->GetActiveCamera()->GetPosition()[0] << " "
           << renderer->GetActiveCamera()->GetPosition()[1] << " "
           << renderer->GetActiveCamera()->GetPosition()[2] << "\n";
    }
};
**/

int main()
{
  // Read in data file

  char filename[256];

  strcpy(filename, "test"); //set filename

  readme(filename); //read in data

  // The pipeline creation is documented in Step1
  //
  //h = 10;
  //r = 2;
  //res = 100;

  vtkConeSource *cone = vtkConeSource::New();
  cone->SetHeight( h );
```

```
cone->SetRadius( r );
cone->SetResolution( res );
//cone->GetProperty()->1.0, 0.0, 0.0);

vtkPolyDataMapper *coneMapper = vtkPolyDataMapper::New();
coneMapper->SetInputConnection( cone->GetOutputPort() );
vtkActor *coneActor = vtkActor::New();
coneActor->SetMapper( coneMapper );

vtkRenderer *ren1= vtkRenderer::New();
ren1->AddActor( coneActor );
ren1->SetBackground( 0.0, 0.0, 1.0 );
ren1->ResetCamera();

vtkRenderWindow *renWin = vtkRenderWindow::New();
renWin->AddRenderer( ren1 );
renWin->SetSize( 500, 500 );

// Here is where we setup the observer, we do a new and ren1 will
// eventually free the observer
/**
vtkMyCallback *mo1 = vtkMyCallback::New();
ren1->AddObserver(vtkCommand::StartEvent,mo1);
mo1->Delete();
**/

//
// now we loop over 360 degrees and render the cone each time
//
int i;
for (i = 0; i < 3600; ++i)
  {
  // render the image
  renWin->Render();
  // rotate the active camera by one degree
  ren1->GetActiveCamera()->Azimuth( 0.1 );
  }

cout << r <<" " << h << " "<< res;

//
// Free up any objects we created
//
cone->Delete();
coneMapper->Delete();
coneActor->Delete();
```

```
  ren1->Delete();
  //renWin->Delete();

  return 0;
}
```

# APPENDIX II –READ NC3D ALPHA

```
/*=================================================
==========================
The first step in creating a self contained VTK Visualization component
for use in the CASELab.  Create single element and rotate it.
=================================================
==========================*/

// first include the required header files for the vtk classes we are using
#include "vtkPolyDataMapper.h"
#include "vtkRenderWindow.h"
#include "vtkCommand.h"
#include "vtkCamera.h"
#include "vtkActor.h"
#include "vtkRenderer.h"
#include <stdio.h>
#include <string.h>

#include "vtkPoints.h" //File for Coord
#include "vtkUnstructuredGrid.h" //File for Unstructured grid
#include "vtkDataSetMapper.h" // use for unstructured grid
//Tetra goes to Cell3D goes to Cell goes to object
#include "vtkIdList.h"
#include "vtkProperty.h"

#include "NetCDF\NetCDF.h" //to use nc commands

//Globals
        int nnd, nel, nsg;
        int nbe, nwl, nsd, nsf;
        double coor[3][4];
        int ielm[4][4], iseg[2][6], ibel[4][48], lbe[40];

//Used as a check
void ncstatus(int status)
{
   if(status != NC_NOERR){
   fprintf(stderr, "%s\n", nc_strerror(status));
   exit(-1);
   }
}

//Read in data from file
void readme(char *name)
{
        int ncid;
        int nndVar, nelVar, nsgVar;
        int nbeVar, nwlVar, nsdVar, nsfVar;
        int coorVar;
        int ielmVar, isegVar, ibelVar, lbeVar;

        char filename[256];
        strcpy(filename,name);
        strcat(filename, ".nc3d");

        ncstatus(nc_open(filename,0,&ncid));

        ncstatus(nc_inq_dimid(ncid, "nnd", &nndVar));
        ncstatus(nc_inq_dimid(ncid, "nel", &nelVar));
        ncstatus(nc_inq_dimid(ncid, "nsg", &nsgVar));
        ncstatus(nc_inq_dimid(ncid, "nbe", &nbeVar));
        //ncstatus(nc_inq_dimid(ncid, "nwl", &nwlVar));
        //ncstatus(nc_inq_dimid(ncid, "nsd", &nsdVar));
        ncstatus(nc_inq_dimid(ncid, "nsf", &nsfVar));

        ncstatus(nc_inq_dimlen(ncid, nndVar, (size_t *) &nnd));
    ncstatus(nc_inq_dimlen(ncid, nelVar, (size_t *) &nel));
    ncstatus(nc_inq_dimlen(ncid, nsgVar, (size_t *) &nsg));
        ncstatus(nc_inq_dimlen(ncid, nbeVar, (size_t *) &nbe));
    //ncstatus(nc_inq_dimlen(ncid, nwlVar, (size_t *) &nwl));
    //ncstatus(nc_inq_dimlen(ncid, nsdVar, (size_t *) &nsd));
    ncstatus(nc_inq_dimlen(ncid, nsfVar, (size_t *) &nsf));

        ncstatus(nc_inq_varid(ncid, "COOR", &coorVar));
    ncstatus(nc_inq_varid(ncid, "IELM", &ielmVar));
    ncstatus(nc_inq_varid(ncid, "ISEG", &isegVar));
    ncstatus(nc_inq_varid(ncid, "IBEL", &ibelVar));
    ncstatus(nc_inq_varid(ncid, "LBE",  &lbeVar));

        ncstatus(nc_get_var_double(ncid, coorVar, &coor[0][0]));
        ncstatus(nc_get_var_int(ncid, ielmVar, &ielm[0][0]));
        ncstatus(nc_get_var_int(ncid, isegVar, &iseg[0][0]));
        ncstatus(nc_get_var_int(ncid, ibelVar, &ibel[0][0]));
        ncstatus(nc_get_var_int(ncid, lbeVar,  &lbe[0]));

        ncstatus(nc_close(ncid));
}

int main()
{
  //Variables
  int j;
  static vtkIdType cells[1][4]={{0,1,2,3}};
```

```
 static vtkIdType List[1][4];

 // Read in data file
 char filename[256];
 strcpy(filename, "test"); //set filename
 readme(filename); //read in data

 /****************************************
   Source Data Manipulation
  ****************************************/
 //Setup points and coords
 vtkPoints *nodes = vtkPoints::New();
 for (j = 0; j<nnd; j++)
 {
         //get points x, y, z
         nodes->InsertPoint(j, coor[0][j], coor[1][j], coor[2][j]);
    cout << "x= " << coor[0][j] << " y= " << coor[1][j] << " z= " <<
coor[2][j] <<endl;
 }

 //Setup unstructured grid
 vtkUnstructuredGrid *element = vtkUnstructuredGrid::New ();  //make grid
 element->Allocate (1000);  //do first

 // Step through elements (cells in vtk)
 //**
 for (int iel = 0; iel < nel; iel++)
   {
         List[0][iel] = ielm[0][iel]-1;
    List[1][iel] = ielm[1][iel]-1;
    List[2][iel] = ielm[2][iel]-1;
    List[3][iel] = ielm[3][iel]-1;
    cout << "L1= " << List[j][0] << " L2= " << List[j][1] << " L3= " <<
List[j][2] << " L4= " << List[j][3]<<endl;
    element->InsertNextCell (10, 4, List[iel]);
   }
 //**/

 //element->InsertNextCell (10, 4, cells[0]); //10 = Tetrahedral, 4=#pts,
cells=list of pts
 element->SetPoints (nodes);
 /****************************************
   Mapper
  ****************************************/
 vtkDataSetMapper *elemMapper = vtkDataSetMapper::New ();
 elemMapper->SetInput (element);
 elemMapper->ImmediateModeRenderingOn();

 /***************************************
   Actor
 ***************************************/
 vtkActor *elemActor = vtkActor::New();
 elemActor->SetMapper( elemMapper );
 elemActor->GetProperty()->SetColor(.8,.8,.8);

 vtkActor *wireActor = vtkActor::New ();
 wireActor->SetMapper (elemMapper);
 wireActor->GetProperty()->SetRepresentationToWireframe();

 //Set up the window
 vtkRenderer *ren1= vtkRenderer::New();
 ren1->AddActor( elemActor );
 ren1->AddActor( wireActor );
 ren1->SetBackground( 1.0, 1.0, 1.0 ); //color: white
 ren1->ResetCamera();

 vtkRenderWindow *renWin = vtkRenderWindow::New();
 renWin->AddRenderer( ren1 );
 renWin->SetSize( 500, 500 );

 // loop over 360 degrees and render the cone each time
 int i;
 for (i = 0; i < 3600; ++i)
   {
   // render the image
   renWin->Render();
   // rotate the active camera by one degree
   ren1->GetActiveCamera()->Azimuth( 0.1 );
   }

 // Free up any objects we created
 element->Delete();
 elemMapper->Delete();
 elemActor->Delete();
 ren1->Delete();
 //renWin->Delete();

 return 0;
}
```

## APPENDIX III –READ NC3D BRAVO

```
/**************************************************

Another step in creating a self contained VTK Visualization component
for use in the CASELab.  Create single element and interact with it

**************************************************/

//for the vtk classes we are using
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkRenderer.h"

#include "vtkPoints.h"
#include "vtkUnstructuredGrid.h"
#include "vtkDataSetMapper.h"
#include "vtkActor.h"
#include "vtkProperty.h"
#include "vtkCamera.h"
#include "NetCDF\NetCDF.h" //to use nc commands


//Globals
        int nnd, nel, nsg;
        int nbe, nwl, nsd, nsf;
        double coor[3][4];
        int ielm[4][1], iseg[2][100], ibel[5][48], lbe[100];

void ncstatus(int status)
{
   if(status != NC_NOERR){
   fprintf(stderr, "%s\n", nc_strerror(status));
   exit(-1);
   }
}

void readme(char *name)
{
        int ncid;
        int nndVar, nelVar, nsgVar;
        int nbeVar, nwlVar, nsdVar, nsfVar;
        int coorVar;
        int ielmVar, isegVar, ibelVar, lbeVar;

        char filename[256];
        strcpy(filename,name);
        strcat(filename, ".nc3d");
```

```
          ncstatus(nc_open(filename,0,&ncid));

          ncstatus(nc_inq_dimid(ncid, "nnd", &nndVar));
          ncstatus(nc_inq_dimid(ncid, "nel", &nelVar));
          ncstatus(nc_inq_dimid(ncid, "nsg", &nsgVar));
          ncstatus(nc_inq_dimid(ncid, "nbe", &nbeVar));
          //ncstatus(nc_inq_dimid(ncid, "nwl", &nwlVar));
          //ncstatus(nc_inq_dimid(ncid, "nsd", &nsdVar));
          ncstatus(nc_inq_dimid(ncid, "nsf", &nsfVar));

          ncstatus(nc_inq_dimlen(ncid, nndVar, (size_t *) &nnd));
   ncstatus(nc_inq_dimlen(ncid, nelVar, (size_t *) &nel));
   ncstatus(nc_inq_dimlen(ncid, nsgVar, (size_t *) &nsg));
          ncstatus(nc_inq_dimlen(ncid, nbeVar, (size_t *) &nbe));
   //ncstatus(nc_inq_dimlen(ncid, nwlVar, (size_t *) &nwl));
   //ncstatus(nc_inq_dimlen(ncid, nsdVar, (size_t *) &nsd));
   ncstatus(nc_inq_dimlen(ncid, nsfVar, (size_t *) &nsf));

          ncstatus(nc_inq_varid(ncid, "COOR", &coorVar));
   ncstatus(nc_inq_varid(ncid, "IELM", &ielmVar));
   ncstatus(nc_inq_varid(ncid, "ISEG", &isegVar));
   ncstatus(nc_inq_varid(ncid, "IBEL", &ibelVar));
   ncstatus(nc_inq_varid(ncid, "LBE",  &lbeVar));

          ncstatus(nc_get_var_double(ncid, coorVar, &coor[0][0]));
          ncstatus(nc_get_var_int(ncid, ielmVar, &ielm[0][0]));
          ncstatus(nc_get_var_int(ncid, isegVar, &iseg[0][0]));
          ncstatus(nc_get_var_int(ncid, ibelVar, &ibel[0][0]));
          ncstatus(nc_get_var_int(ncid, lbeVar,  &lbe[0]));

          ncstatus(nc_close(ncid));
}

int main( int argc, char *argv[] )
{
  //Variables
  int i;
  int j;

  // Read in data file
  char filename[256];
  strcpy(filename, "test"); //set filename
  readme(filename); //read in data

  cout << "nnd = " << nnd << endl;
  cout << "nel = " << nel << endl;
  cout << "nbe = " << nbe << endl;
  static vtkIdType nList[4][1];
  //static vtkIdType sol[48][3];

  /****************************************
    Source Data Manipulation
  ****************************************/
  //Setup points and coords
  vtkPoints *nodes = vtkPoints::New();
  for (j = 0; j<nnd; j++)
  {
            //get points x, y, z
            nodes->InsertPoint(j, coor[0][j], coor[1][j], coor[2][j]);
     //cout << "x= " << coor[0][j] << " y= " << coor[1][j] << " z= " <<
coor[2][j] <<endl;
  }

  //Setup unstructured grid
  vtkUnstructuredGrid *element = vtkUnstructuredGrid::New (); //make grid
  element->Allocate (1000); //do first

  //vtkUnstructuredGrid *boundary = vtkUnstructuredGrid::New (); //make grid
  //boundary->Allocate (1000); //do first

  // Step through elements (cells in vtk)

  for (int iel = 0; iel < nel; iel++)
  {
    nList[0][iel] = ielm[0][iel]-1;
    nList[1][iel] = ielm[1][iel]-1;
    nList[2][iel] = ielm[2][iel]-1;
    nList[3][iel] = ielm[3][iel]-1;
    cout << "L1= " << nList[iel][0] << " L2= " << nList[iel][1] << " L3= " <<
nList[iel][2] << " L4= " << nList[iel][3]<<endl;
    element->InsertNextCell (10, 4, nList[iel]);
  }
/**
  for (int ibe = 0; ibe < nbe; ibe++)
  {
    sol[0][ibe] = ibel[ibe][0];
    sol[1][ibe] = ibel[ibe][1];
    sol[2][ibe] = ibel[ibe][2];
    boundary->InsertNextCell (5, 3, sol[iel]);
  }
**/
```

```
 //element->InsertNextCell (10, 4, cells[0]); //10 = Tetrahedral, 4=#pts,
cells=list of pts
 element->SetPoints (nodes);
 //boundary->SetPoints (nodes);
 nodes->Delete();

/************************************
 Mappers
************************************/
 vtkDataSetMapper *elementMapper = vtkDataSetMapper::New();
    elementMapper->SetInput(element);
    elementMapper->ImmediateModeRenderingOn();
 //vtkDataSetMapper *boundaryMapper = vtkDataSetMapper::New();
    //boundaryMapper->SetInput(boundary);
    //boundaryMapper->ImmediateModeRenderingOn();

/************************************
 Actors
************************************/
 vtkActor *elementActor = vtkActor::New();
    elementActor->SetMapper(elementMapper);
    elementActor->GetProperty()->SetColor(.8,.8,.8);
    elementActor->AddPosition(0,0.001,0);
 //vtkActor *boundActor = vtkActor::New();
    //boundActor->SetMapper(boundaryMapper);
    //boundActor->GetProperty()->SetColor(.8,.8,.8);
    //boundActor->AddPosition(0,0.001,0);
 vtkActor *wireActor = vtkActor::New();
    wireActor->SetMapper(elementMapper);
    wireActor->GetProperty()->SetRepresentationToWireframe();
    wireActor->GetProperty()->SetColor(0,0,0);

/************************************
 Render Window
************************************/
 vtkRenderer *renderer = vtkRenderer::New();
 vtkRenderWindow *renWin = vtkRenderWindow::New();
   renWin->AddRenderer(renderer);
 vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
   iren->SetRenderWindow(renWin);

 renderer->AddActor(elementActor);
           //renderer->AddActor(boundActor);
    renderer->AddActor(wireActor);
    renderer->SetBackground(1,1,1);
    renderer->GetActiveCamera()->Elevation(60.0);
    renderer->GetActiveCamera()->Azimuth(30.0);
    renderer->GetActiveCamera()->Zoom(0);

 renWin->SetSize(300,300);

 // interact with data
 renWin->Render();

 //SAVEIMAGE( renWin );

 iren->Start();

 // Clean up
 renderer->Delete();
 renWin->Delete();
 iren->Delete();
 nodes->Delete();
 element->Delete();
 elementMapper->Delete();
 elementActor->Delete();
 wireActor->Delete();

 return 0;
}
```

## APPENDIX IV –READ NC3D CHARLIE

```
/************************************************
Cody Pinkerman
nc3d_Charlie

Read in single element, and show boundary elements
  especially solid wall elements.

************************************************/

//for the vtk classes we are using
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkRenderer.h"

#include "vtkPoints.h"
#include "vtkUnstructuredGrid.h"
#include "vtkDataSetMapper.h"
#include "vtkActor.h"
#include "vtkProperty.h"
#include "vtkCamera.h"
#include "NetCDF\NetCDF.h" //to use nc commands


//Globals
        int nnd, nel, nsg;
        int nbe, nwl, nsd, nsf;
        double coor[3][4]; //[3][nnd]
        int ielm[4][1]; //[4][nel]
        int iseg[2][6]; //[2][nsg]
        int ibel[5][4]; //[5][nbe]
        int lbe[8];

void ncstatus(int status)
{
   if(status != NC_NOERR){
   fprintf(stderr, "%s\n", nc_strerror(status));
   exit(-1);
   }
}

void readme(char *name)
{
        int ncid;
        int nndVar, nelVar, nsgVar;
        int nbeVar, nwlVar, nsdVar, nsfVar;
        int coorVar;
        int ielmVar, isegVar, ibelVar, lbeVar;

        char filename[256];
        strcpy(filename,name);
        strcat(filename, ".nc3d");

        ncstatus(nc_open(filename,0,&ncid));

        ncstatus(nc_inq_dimid(ncid, "nnd", &nndVar));
        ncstatus(nc_inq_dimid(ncid, "nel", &nelVar));
        ncstatus(nc_inq_dimid(ncid, "nsg", &nsgVar));
        ncstatus(nc_inq_dimid(ncid, "nbe", &nbeVar));
        //ncstatus(nc_inq_dimid(ncid, "nwl", &nwlVar));
        //ncstatus(nc_inq_dimid(ncid, "nsd", &nsdVar));
        ncstatus(nc_inq_dimid(ncid, "nsf", &nsfVar));

        ncstatus(nc_inq_dimlen(ncid, nndVar, (size_t *) &nnd));
    ncstatus(nc_inq_dimlen(ncid, nelVar, (size_t *) &nel));
    ncstatus(nc_inq_dimlen(ncid, nsgVar, (size_t *) &nsg));
        ncstatus(nc_inq_dimlen(ncid, nbeVar, (size_t *) &nbe));
    //ncstatus(nc_inq_dimlen(ncid, nwlVar, (size_t *) &nwl));
    //ncstatus(nc_inq_dimlen(ncid, nsdVar, (size_t *) &nsd));
    ncstatus(nc_inq_dimlen(ncid, nsfVar, (size_t *) &nsf));

        ncstatus(nc_inq_varid(ncid, "COOR", &coorVar));
    ncstatus(nc_inq_varid(ncid, "IELM", &ielmVar));
    ncstatus(nc_inq_varid(ncid, "ISEG", &isegVar));
    ncstatus(nc_inq_varid(ncid, "IBEL", &ibelVar));
    ncstatus(nc_inq_varid(ncid, "LBE",  &lbeVar));

        ncstatus(nc_get_var_double(ncid, coorVar, &coor[0][0]));
        ncstatus(nc_get_var_int(ncid, ielmVar, &ielm[0][0]));
        ncstatus(nc_get_var_int(ncid, isegVar, &iseg[0][0]));
        ncstatus(nc_get_var_int(ncid, ibelVar, &ibel[0][0]));
        ncstatus(nc_get_var_int(ncid, lbeVar,  &lbe[0]));

        ncstatus(nc_close(ncid));
}

int main( int argc, char *argv[] )
{
  //Variables
  int i;
  int j;

  // Read in data file
  char filename[256];
  cout << "Please enter file name " <<endl;
  cin >> filename;

  //strcpy(filename, "test"); //set filename
  readme(filename); //read in data

  cout << "nnd = " << nnd << endl;
  cout << "nel = " << nel << endl;
  cout << "nsg = " << nsg << endl;
  cout << "nbe = " << nbe << endl;

  static vtkIdType nList[4];
  static vtkIdType sol[3];


  /****************************************
    Source Data Manipulation
  ****************************************/
  //Setup points and coords
  vtkPoints *nodes = vtkPoints::New();
  for (j = 0; j<nnd; j++)
  {
            //get points x, y, z
            nodes->InsertPoint(j, coor[0][j], coor[1][j], coor[2][j]);
     //cout << "x= " << coor[0][j] << " y= " << coor[1][j] << " z= " <<
coor[2][j] <<endl;
  }

  //Setup unstructured grid
  vtkUnstructuredGrid *element = vtkUnstructuredGrid::New (); //make grid
  element->Allocate (1000); //do first

  vtkUnstructuredGrid *boundary = vtkUnstructuredGrid::New (); //make grid
  boundary->Allocate (1000); //do first

  // Step through elements (cells in vtk)

  for (int iel = 0; iel < nel; iel++)
  {
    nList[0] = ielm[0][iel]-1;
    nList[1] = ielm[1][iel]-1;
    nList[2] = ielm[2][iel]-1;
    nList[3] = ielm[3][iel]-1;
    //cout << "L1= " << nList[0][iel] << " L2= " << nList[1][iel] << " L3= "
<< nList[2][iel] << " L4= " << nList[3][iel]<<endl;
```

```
          element->InsertNextCell (10, 4, nList);
  }

  for (int ibe = 0; ibe < lbe[1]; ibe++)
  {
    sol[0] = ibel[0][ibe]-1;
    sol[1] = ibel[1][ibe]-1;
    sol[2] = ibel[2][ibe]-1;
    //cout << "B1= " << sol[0][ibe] << " B2= " << sol[1][ibe] << " B3= " <<
sol[2][ibe]<<endl;
          //if( sol[3][ibe] == 1)
          //{
          boundary->InsertNextCell (5, 3, sol);
          //}
  }

    //sol[0] = ibel[0][0]-1;
    //sol[1] = ibel[1][0]-1;
    //sol[2] = ibel[2][0]-1;
    //cout << "B1= " << sol[0][0] << " B2= " << sol[1][0] << " B3= " <<
sol[2][0]<<endl;
          //boundary->InsertNextCell (5, 3, sol);

  element->SetPoints (nodes);
  boundary->SetPoints (nodes);
  //nodes->Delete();

/************************************
 Mappers
************************************/
  vtkDataSetMapper *elementMapper = vtkDataSetMapper::New();
     elementMapper->SetInput(element);
     elementMapper->ImmediateModeRenderingOn();
  vtkDataSetMapper *boundaryMapper = vtkDataSetMapper::New();
     boundaryMapper->SetInput(boundary);
     boundaryMapper->ImmediateModeRenderingOn();

/************************************
 Actors
************************************/
  vtkActor *elementActor = vtkActor::New();
     elementActor->SetMapper(elementMapper);
     elementActor->GetProperty()->SetColor(.8,.8,.8); //gray
     elementActor->AddPosition(0,0.001,0);
  vtkActor *boundActor = vtkActor::New();
     boundActor->SetMapper(boundaryMapper);
```

```
      boundActor->GetProperty()->SetColor(0,0,1); //blue
      boundActor->AddPosition(0,0.001,0);
  vtkActor *wireActor = vtkActor::New();
      wireActor->SetMapper(elementMapper);
      wireActor->GetProperty()->SetRepresentationToWireframe();
      wireActor->GetProperty()->SetColor(0,0,0); //black

/************************************
 Render Window
************************************/
  vtkRenderer *renderer = vtkRenderer::New();
  vtkRenderWindow *renWin = vtkRenderWindow::New();
    renWin->AddRenderer(renderer);
  vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
    iren->SetRenderWindow(renWin);

  //renderer->AddActor(elementActor);
            renderer->AddActor(boundActor);
            renderer->AddActor(wireActor);
            renderer->SetBackground(1,1,1);
      renderer->GetActiveCamera()->Elevation(60.0);
      renderer->GetActiveCamera()->Azimuth(30.0);
      renderer->GetActiveCamera()->Zoom(0);

          renWin->SetSize(300,300);

cout << "Point Alpha " <<endl;
  // interact with data
  renWin->Render();
cout << "Point Bravo " <<endl;

  iren->Start();

  // Clean up
/**
  renderer->Delete();
  renWin->Delete();
  iren->Delete();
  nodes->Delete();
  element->Delete();
  elementMapper->Delete();
  elementActor->Delete();
  boundary->Delete();
  boundaryMapper->Delete();
  boundActor->Delete();
  wireActor->Delete();
```

```
**/

  return 0;
}
```

## APPENDIX V –READ NC3D DELTA

```
/***********************************************
Cody Pinkerman
nc3d_Delta
VTK 5.0.4

Read in multiple element, and show boundary elements
  and solid walls.

***********************************************/

//for the vtk classes we are using
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkRenderer.h"

#include "vtkPoints.h"
#include "vtkUnstructuredGrid.h"
#include "vtkDataSetMapper.h"
#include "vtkActor.h"
#include "vtkProperty.h"
#include "vtkCamera.h"
#include "NetCDF\NetCDF.h" //to use nc commands

//Globals
	int nnd, nel, nsg;
	int nbe, nwl, nsd, nsf;

	/** Box
	double coor[3][31]; //[3][nnd]
	int ielm[4][72]; //[4][nel]
	int iseg[2][126]; //[2][nsg]
	int ibel[5][48]; //[5][nbe]
	//**/
	/** Element
	double coor[3][4]; //[3][nnd]
	int ielm[4][1]; //[4][nel]
	int iseg[2][6]; //[2][nsg]
	int ibel[5][4]; //[5][nbe]
	/**/
	/** Lemon
	double coor[3][5663]; //[3][nnd]
	int ielm[4][27440]; //[4][nel]
	int iseg[2][34797]; //[2][nsg]
	int ibel[5][3390]; //[5][nbe]
	/**/
	/** Double Wing
```

```
        double coor[3][4101]; //[3][nnd]
        int ielm[4][20158]; //[4][nel]
        int iseg[2][25428]; //[2][nsg]
        int ibel[5][2338]; //[5][nbe]
        /**/
        /** Missle
        double coor[3][854738]; //[3][nnd]
        int ielm[4][149885]; //[4][nel]
        int iseg[2][1005511]; //[2][nsg]
        int ibel[5][1780]; //[5][nbe]
        /**/
        //** f22
        double coor[3][462216]; //[3][nnd]
        int ielm[4][2517507]; //[4][nel]
        int iseg[2][3029220]; //[2][nsg]
        int ibel[5][98998]; //[5][nbe]
        /**/
        int lbe[8];

void ncstatus(int status)
{
   if(status != NC_NOERR){
   fprintf(stderr, "%s\n", nc_strerror(status));
   exit(-1);
   }
}

void readme(char *name)
{
        int ncid;
        int nndVar, nelVar, nsgVar;
        int nbeVar, nwlVar, nsdVar, nsfVar;
        int coorVar;
        int ielmVar, isegVar, ibelVar, lbeVar;

        char filename[256];
        strcpy(filename,name);
        strcat(filename, ".nc3d");

        ncstatus(nc_open(filename,0,&ncid));

        ncstatus(nc_inq_dimid(ncid, "nnd", &nndVar));
        ncstatus(nc_inq_dimid(ncid, "nel", &nelVar));
        ncstatus(nc_inq_dimid(ncid, "nsg", &nsgVar));
        ncstatus(nc_inq_dimid(ncid, "nbe", &nbeVar));
        //ncstatus(nc_inq_dimid(ncid, "nwl", &nwlVar));
        //ncstatus(nc_inq_dimid(ncid, "nsd", &nsdVar));
        ncstatus(nc_inq_dimid(ncid, "nsf", &nsfVar));

        ncstatus(nc_inq_dimlen(ncid, nndVar, (size_t *) &nnd));
   ncstatus(nc_inq_dimlen(ncid, nelVar, (size_t *) &nel));
   ncstatus(nc_inq_dimlen(ncid, nsgVar, (size_t *) &nsg));
        ncstatus(nc_inq_dimlen(ncid, nbeVar, (size_t *) &nbe));
   //ncstatus(nc_inq_dimlen(ncid, nwlVar, (size_t *) &nwl));
   //ncstatus(nc_inq_dimlen(ncid, nsdVar, (size_t *) &nsd));
   ncstatus(nc_inq_dimlen(ncid, nsfVar, (size_t *) &nsf));

        ncstatus(nc_inq_varid(ncid, "COOR", &coorVar));
   ncstatus(nc_inq_varid(ncid, "IELM", &ielmVar));
   ncstatus(nc_inq_varid(ncid, "ISEG", &isegVar));
   ncstatus(nc_inq_varid(ncid, "IBEL", &ibelVar));
   ncstatus(nc_inq_varid(ncid, "LBE",  &lbeVar));

        ncstatus(nc_get_var_double(ncid, coorVar, &coor[0][0]));
        ncstatus(nc_get_var_int(ncid, ielmVar, &ielm[0][0]));
        ncstatus(nc_get_var_int(ncid, isegVar, &iseg[0][0]));
        ncstatus(nc_get_var_int(ncid, ibelVar, &ibel[0][0]));
        ncstatus(nc_get_var_int(ncid, lbeVar,  &lbe[0]));

        ncstatus(nc_close(ncid));

}

int main( int argc, char *argv[] )
{
 //Variables
 int i;
 int j;

 // Read in data file
 char filename[256];
 cout << "Please enter file name " <<endl;
 cin >> filename;

 readme(filename); //read in data

 cout << "nnd = " << nnd << endl;
 cout << "nel = " << nel << endl;
 cout << "nsg = " << nsg << endl;
 cout << "nbe = " << nbe << endl;

 static vtkIdType npts[4]; //nodal points
 static vtkIdType sol[3]; //solid surfaces

 /****************************************
   Source Data Manipulation
 ****************************************/
 //Setup points and coords
 vtkPoints *nodes = vtkPoints::New();
 for (j = 0; j<nnd; j++)
 {
         //get points x, y, z
         nodes->InsertPoint(j, coor[0][j], coor[1][j], coor[2][j]);
 }

 //Setup unstructured grid
 vtkUnstructuredGrid *element = vtkUnstructuredGrid::New ();  //make grid
 element->Allocate (1000);  //do first

 vtkUnstructuredGrid *boundary = vtkUnstructuredGrid::New ();  //make grid
 boundary->Allocate (1000);  //do first

 //cout <<"-------------------------------------- "<<endl;

 // Step through elements (cells in vtk)
 for (int iel = 0; iel < nel; iel++)
 {
   npts[0] = ielm[0][iel]-1;
   npts[1] = ielm[1][iel]-1;
   npts[2] = ielm[2][iel]-1;
   npts[3] = ielm[3][iel]-1;
        element->InsertNextCell (10, 4, npts);
 }

 for (int ibe = 0; ibe < lbe[1]; ibe++)
 {
   sol[0] = ibel[0][ibe]-1;
   sol[1] = ibel[1][ibe]-1;
   sol[2] = ibel[2][ibe]-1;
        boundary->InsertNextCell (5, 3, sol);
 }

 element->SetPoints (nodes);
 boundary->SetPoints (nodes);
 nodes->Delete();

/************************************
 Mappers
```

```
***********************************/
 vtkDataSetMapper *elementMapper = vtkDataSetMapper::New();
    elementMapper->SetInput(element);
    elementMapper->ImmediateModeRenderingOn();
 vtkDataSetMapper *boundaryMapper = vtkDataSetMapper::New();
    boundaryMapper->SetInput(boundary);
    boundaryMapper->ImmediateModeRenderingOn();

/***********************************
 Actors
***********************************/
 vtkActor *elementActor = vtkActor::New();
    elementActor->SetMapper(elementMapper);
    elementActor->GetProperty()->SetColor(.8,.8,.8); //gray
    elementActor->AddPosition(0,0.001,0);
 vtkActor *boundActor = vtkActor::New();
    boundActor->SetMapper(boundaryMapper);
    boundActor->GetProperty()->SetColor(0,0,1); //blue
    boundActor->AddPosition(0,0.001,0);
 vtkActor *wireActor = vtkActor::New();
    wireActor->SetMapper(elementMapper);
    wireActor->GetProperty()->SetRepresentationToWireframe();
    wireActor->GetProperty()->SetColor(0,0,0); //black

/***********************************
 Render Window
***********************************/
 vtkRenderer *renderer = vtkRenderer::New();
 vtkRenderWindow *renWin = vtkRenderWindow::New();
   renWin->AddRenderer(renderer);
 vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
   iren->SetRenderWindow(renWin);

 //renderer->AddActor(elementActor);
          renderer->AddActor(boundActor);
          renderer->AddActor(wireActor);
          renderer->SetBackground(1,1,1);
    renderer->GetActiveCamera()->Elevation(60.0);
    renderer->GetActiveCamera()->Azimuth(30.0);
    renderer->GetActiveCamera()->Zoom(0);

         renWin->SetSize(500,500);

 cout << "Point Alpha " <<endl;
 // interact with data
 renWin->Render();
```

```
  cout << "Point Bravo " <<endl;

  iren->Start();

  // Clean up
  renderer->Delete();
  renWin->Delete();
  //iren->Delete();  //commented to stop error when closing
  nodes->Delete();
  element->Delete();
  elementMapper->Delete();
  elementActor->Delete();
  boundary->Delete();
  boundaryMapper->Delete();
  boundActor->Delete();
  wireActor->Delete();

  return 0;
}
```

## APPENDIX VI –READ NC3D ECHO

```
/**********************************************
Cody Pinkerman
nc3d_Echo
VTK 5.0.4

Read in multiple element, and show boundary elements
  and solid walls.
Added .unk file reader for density.
Along with ability to plot it (maybe).

**********************************************/

//for the vtk classes we are using
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkRenderer.h"

#include "vtkPoints.h"
#include "vtkUnstructuredGrid.h"
#include "vtkDataSetMapper.h"
#include "vtkActor.h"
#include "vtkProperty.h"
#include "vtkCamera.h"
#include "vtkScalarBarActor.h"
#include "vtkLookupTable.h"
#include "vtkPointData.h"
#include "vtkFloatArray.h"
#include "NetCDF\NetCDF.h" //to use nc commands

//Globals
        int nnd, nel, nsg;
        int nbe, nwl, nsd, nsf;

        /** Box
        double coor[3][31]; //[3][nnd]
        int ielm[4][72]; //[4][nel]
        int iseg[2][126]; //[2][nsg]
        int ibel[5][48]; //[5][nbe]
        double density[31]; //[nnd]
        //**/
        /** Element
        double coor[3][4]; //[3][nnd]
        int ielm[4][1]; //[4][nel]
        int iseg[2][6]; //[2][nsg]
        int ibel[5][4]; //[5][nbe]
        double density[4]; //[nnd]
        /**/
        /** Lemon
        double coor[3][5663]; //[3][nnd]
        int ielm[4][27440]; //[4][nel]
        int iseg[2][34797]; //[2][nsg]
        int ibel[5][3390]; //[5][nbe]
        double density[5663]; //[nnd]
        /**/
        /** Double Wing
        double coor[3][4101]; //[3][nnd]
        int ielm[4][20158]; //[4][nel]
        int iseg[2][25428]; //[2][nsg]
        int ibel[5][2338]; //[5][nbe]
        double density[4101]; //[nnd]
        /**/
        /** Missle
        double coor[3][854738]; //[3][nnd]
        int ielm[4][149885]; //[4][nel]
        int iseg[2][1005511]; //[2][nsg]
        int ibel[5][1780]; //[5][nbe]
        double density[854738]; //[nnd]
        /**/
        /** f22
        double coor[3][462216]; //[3][nnd]
        int ielm[4][2517507]; //[4][nel]
        int iseg[2][3029220]; //[2][nsg]
        int ibel[5][98998]; //[5][nbe]
        double density[462216]; //[nnd]
        /**/
        //** NASA
        double coor[3][381740]; //[3][nnd]
        int ielm[4][2124680]; //[4][nel]
        int iseg[2][2531602]; //[2][nsg]
        int ibel[5][50370]; //[5][nbe]
        double density[381740]; //[nnd]
        /**/
        /** NASA2
        double coor[3][122288]; //[3][nnd]
        int ielm[4][665109]; //[4][nel]
        int iseg[2][801433]; //[2][nsg]
        int ibel[5][28074]; //[5][nbe]
        double density[122288]; //[nnd]
        /**/
        int lbe[8];

void ncstatus(int status)
```

```
{
   if(status != NC_NOERR){
   fprintf(stderr, "%s\n", nc_strerror(status));
   exit(-1);
   }
}

void readgeom(char *name)
{
         int ncid;
         int nndVar, nelVar, nsgVar;
         int nbeVar, nwlVar, nsdVar, nsfVar;
         int coorVar;
         int ielmVar, isegVar, ibelVar, lbeVar;

         char filename[256];
         strcpy(filename,name);
         strcat(filename, ".nc3d");

         ncstatus(nc_open(filename,0,&ncid));

         ncstatus(nc_inq_dimid(ncid, "nnd", &nndVar));
         ncstatus(nc_inq_dimid(ncid, "nel", &nelVar));
         ncstatus(nc_inq_dimid(ncid, "nsg", &nsgVar));
         ncstatus(nc_inq_dimid(ncid, "nbe", &nbeVar));
         //ncstatus(nc_inq_dimid(ncid, "nwl", &nwlVar));
         //ncstatus(nc_inq_dimid(ncid, "nsd", &nsdVar));
         ncstatus(nc_inq_dimid(ncid, "nsf", &nsfVar));

         ncstatus(nc_inq_dimlen(ncid, nndVar, (size_t *) &nnd));
   ncstatus(nc_inq_dimlen(ncid, nelVar, (size_t *) &nel));
   ncstatus(nc_inq_dimlen(ncid, nsgVar, (size_t *) &nsg));
         ncstatus(nc_inq_dimlen(ncid, nbeVar, (size_t *) &nbe));
   //ncstatus(nc_inq_dimlen(ncid, nwlVar, (size_t *) &nwl));
   //ncstatus(nc_inq_dimlen(ncid, nsdVar, (size_t *) &nsd));
   ncstatus(nc_inq_dimlen(ncid, nsfVar, (size_t *) &nsf));

         ncstatus(nc_inq_varid(ncid, "COOR", &coorVar));
   ncstatus(nc_inq_varid(ncid, "IELM", &ielmVar));
   ncstatus(nc_inq_varid(ncid, "ISEG", &isegVar));
   ncstatus(nc_inq_varid(ncid, "IBEL", &ibelVar));
   ncstatus(nc_inq_varid(ncid, "LBE",  &lbeVar));

         ncstatus(nc_get_var_double(ncid, coorVar, &coor[0][0]));
         ncstatus(nc_get_var_int(ncid, ielmVar, &ielm[0][0]));
         ncstatus(nc_get_var_int(ncid, isegVar, &iseg[0][0]));
```

```
        ncstatus(nc_get_var_int(ncid, ibelVar, &ibel[0][0]));
        ncstatus(nc_get_var_int(ncid, lbeVar,  &lbe[0]));

        ncstatus(nc_close(ncid));

}

void readunk(char *name)
{
        int ncid;
        int rhoVar;

        char filename[256];
        strcpy(filename,name);
        strcat(filename, ".unk");

        ncstatus(nc_open(filename,0,&ncid));

        ncstatus(nc_inq_varid(ncid, "Density", &rhoVar));

        ncstatus(nc_get_var_double(ncid, rhoVar,  &density[0]));

        ncstatus(nc_close(ncid));
}

int main( int argc, char *argv[] )
{
  //Variables
  int i;
  int j;
  static vtkIdType thenode[1]; //nodes
  static vtkIdType npts[4]; //nodal info for elements
  static vtkIdType sol[3]; //solid surfaces

  // Read in data file
  char filename[256];
  cout << "Please enter file name " <<endl;
  cin >> filename;

  readgeom(filename); //read in geometry
  readunk(filename); //read in unk

  //output for testing
  cout << "nnd = " << nnd << endl;
  cout << "nel = " << nel << endl;
  cout << "nsg = " << nsg << endl;
  cout << "nbe = " << nbe << endl;

  /****************************************
    Source Data Manipulation
  ****************************************/
  //Setup unstructured grids
  vtkUnstructuredGrid *node = vtkUnstructuredGrid::New ();  //make nodes
  node->Allocate (1000);  //do first

  vtkUnstructuredGrid *element = vtkUnstructuredGrid::New ();  //make grid
  element->Allocate (1000);  //do first

  vtkUnstructuredGrid *boundary = vtkUnstructuredGrid::New ();  //make solids
surface
  boundary->Allocate (1000);  //do first

  //Setup nodes and coords (vertex (1) in vtk)
  vtkPoints *nodes = vtkPoints::New();
  for (j = 0; j < nnd; j++)
  {
		//get points x, y, z
		nodes->InsertPoint(j, coor[0][j], coor[1][j], coor[2][j]);
		//to be able to input actual nodes in render
		thenode[0] = j;
		node->InsertNextCell (1, 1, thenode);
  }

  /**Testing for element density
  for (i = 0; i < nnd; i++)
  {
		density[i] = coor[0][i];
  }
  /**/

  // Step through elements (tetra (10) in vtk)
  for (int iel = 0; iel < nel; iel++)
  {
    npts[0] = ielm[0][iel]-1;
    npts[1] = ielm[1][iel]-1;
    npts[2] = ielm[2][iel]-1;
    npts[3] = ielm[3][iel]-1;
		element->InsertNextCell (10, 4, npts);
  }
  // Step through solid surfaces (tri (5) in vtk)
  for (int ibe = 0; ibe < lbe[1]; ibe++)
  {
   sol[0] = ibel[0][ibe]-1;
   sol[1] = ibel[1][ibe]-1;
   sol[2] = ibel[2][ibe]-1;
         boundary->InsertNextCell (5, 3, sol);
  }
  node->SetPoints (nodes);
  element->SetPoints (nodes);
  boundary->SetPoints (nodes);

  //Add if here
  vtkFloatArray *newdensity = vtkFloatArray::New ();
  for (int inode = 0; inode < nnd; inode++)
  {
     newdensity->InsertValue (inode, density[inode]);
  }
  //input data
  node->GetPointData()->SetScalars (newdensity);
  element->GetPointData()->SetScalars (newdensity);
  boundary->GetPointData()->SetScalars (newdensity);
  nodes->Delete();

/************************************
 Mappers
************************************/
  //node mapper
  vtkDataSetMapper *nodeMapper = vtkDataSetMapper::New();
     nodeMapper->SetInput(node);
     nodeMapper->ImmediateModeRenderingOn();

  //elements
  vtkDataSetMapper *elementMapper = vtkDataSetMapper::New();
     elementMapper->SetInput(element);
     elementMapper->ImmediateModeRenderingOn();

  //solid boundary
  vtkDataSetMapper *boundaryMapper = vtkDataSetMapper::New();
     boundaryMapper->SetInput(boundary);
     boundaryMapper->ImmediateModeRenderingOn();
/************************************
 Actors
************************************/
  //nodes
  vtkActor *nodeActor = vtkActor::New();
     nodeActor->SetMapper(nodeMapper);
     nodeActor->GetProperty()->SetColor(0,0,0); //black
           nodeActor->GetProperty()->SetPointSize(2.0);
     nodeActor->AddPosition(0,0.001,0);

  //element
  vtkActor *elementActor = vtkActor::New();
     elementActor->SetMapper(elementMapper);
     //elementActor->GetProperty()->SetColor(.8,.8,.8); //gray
     elementActor->AddPosition(0,0.001,0);

  //solid surface
  vtkActor *boundActor = vtkActor::New();
     boundActor->SetMapper(boundaryMapper);
     //boundActor->GetProperty()->SetColor(0,0,1); //blue
     boundActor->AddPosition(0,0.001,0);
          //boundActor->GetProperty()->SetRepresentationToWireframe();

  //wire frame from elements
  vtkActor *wireActor = vtkActor::New();
     wireActor->SetMapper(elementMapper);
          wireActor->GetProperty()->SetRepresentationToWireframe();
     wireActor->GetProperty()->SetColor(0,0,0); //black

  // Now create a lookup table that consists of the full hue circle
  vtkLookupTable *hueLut = vtkLookupTable::New ();
          hueLut->SetTableRange (0, 1);
          hueLut->SetHueRange (.75,0); // 240/360=blue 270/360=some purple
          hueLut->SetSaturationRange (1, 1);
          hueLut->SetValueRange (1, 1);
          hueLut->Build ();     //effective built

  //Color Mapping
  vtkDataSetMapper *datamapper = vtkDataSetMapper::New();
          datamapper->SetInput(boundary);
          datamapper->SetScalarRange(boundary->GetScalarRange());
     datamapper->SetLookupTable(hueLut);
  vtkActor *dataActor = vtkActor::New();
          dataActor->SetMapper(datamapper);

  //Scalar color bar
  vtkScalarBarActor *scalarBar = vtkScalarBarActor::New();
          //scalarBar->SetLookupTable(datamapper->GetLookupTable());
          scalarBar->SetOrientationToVertical();
          scalarBar->SetTitle("Density");
          scalarBar->SetNumberOfLabels(2);
          scalarBar->SetLookupTable(hueLut);

/************************************
```

```
 Render Window
***********************************/
  vtkRenderer *renderer = vtkRenderer::New();
  vtkRenderWindow *renWin = vtkRenderWindow::New();
    renWin->AddRenderer(renderer);
  vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
    iren->SetRenderWindow(renWin);

  //renderer->AddActor(elementActor);
            //renderer->AddActor(nodeActor);
            //renderer->AddActor(boundActor);
            //renderer->AddActor(wireActor);
            renderer->AddActor(scalarBar);
            renderer->AddActor(dataActor);
            renderer->SetBackground(1,1,1);
     renderer->GetActiveCamera()->Elevation(60.0);
     renderer->GetActiveCamera()->Azimuth(30.0);
     renderer->GetActiveCamera()->Zoom(0);

          renWin->SetSize(400,400);

  cout << "Point Alpha " <<endl;
  // interact with data
  renWin->Render();
  cout << "Point Bravo " <<endl;

  iren->Start();
  // Clean up
  renderer->Delete();
  renWin->Delete();
  //iren->Delete();  //commented to stop error when closing
  nodes->Delete();
  element->Delete();
  elementMapper->Delete();
  elementActor->Delete();
  boundary->Delete();
  boundaryMapper->Delete();
  boundActor->Delete();
  wireActor->Delete();

  return 0;
}
```

## APPENDIX VII –READ NC3D FOXTROT

```
/************************************************
Cody Pinkerman
nc3d_Foxtrot
VTK 5.0.4

Read in multiple element, and show boundary elements
  and solid walls.
Added .unk file reader for density, pressure or velocity mag.
Along with ability to plot it.

Update:
Added other variables (entropy, enthalpy, Mach, internal and total energy).
Along with a loop to keep the program active so one can change between
variables.
Also added the ability to plot symetry planes.  These are added to the boundary
grid.

************************************************/

//for the vtk classes we are using
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkRenderer.h"

#include "vtkPoints.h"
#include "vtkUnstructuredGrid.h"
#include "vtkDataSetMapper.h"
#include "vtkActor.h"
#include "vtkProperty.h"
#include "vtkCamera.h"
#include "vtkScalarBarActor.h"
#include "vtkLookupTable.h"
#include "vtkPointData.h"
#include "vtkFloatArray.h"
#include "vtkTextProperty.h"
#include "NetCDF\NetCDF.h" //to use nc commands

//Globals
	int nnd, nel, nsg;
	int nbe, nwl, nsd, nsf;

	/** Box
	double coor[3][31]; //[3][nnd]
	int ielm[4][72]; //[4][nel]
	int iseg[2][126]; //[2][nsg]
	int ibel[5][48]; //[5][nbe]
```

```
double density[31]; //[nnd]
double pressure[31]; //[nnd]
//**/
/** Element
double coor[3][4]; //[3][nnd]
int ielm[4][1]; //[4][nel]
int iseg[2][6]; //[2][nsg]
int ibel[5][4]; //[5][nbe]
double density[4]; //[nnd]
double pressure[4]; //[nnd]
/**/
/** Lemon
double coor[3][5663]; //[3][nnd]
int ielm[4][27440]; //[4][nel]
int iseg[2][34797]; //[2][nsg]
int ibel[5][3390]; //[5][nbe]
double density[5663]; //[nnd]
double pressure[5663]; //[nnd]
/**/
/** Double Wing
double coor[3][4101]; //[3][nnd]
int ielm[4][20158]; //[4][nel]
int iseg[2][25428]; //[2][nsg]
int ibel[5][2338]; //[5][nbe]
double density[4101]; //[nnd]
double pressure[4101]; //[nnd]
/**/
/** Missle
double coor[3][854738]; //[3][nnd]
int ielm[4][149885]; //[4][nel]
int iseg[2][1005511]; //[2][nsg]
int ibel[5][1780]; //[5][nbe]
double density[854738]; //[nnd]
double pressure[854738]; //[nnd]
/**/
/** compr3d
double coor[3][119195]; //[3][nnd]
int ielm[4][649637]; //[4][nel]
int iseg[2][780442]; //[2][nsg]
int ibel[5][23222]; //[5][nbe]
double density[119195]; //[nnd]
double pressure[119195]; //[nnd]
double Vx[119195], Vy[119195], Vz[119195];//[nnd]
/**/
/** f22
double coor[3][462216]; //[3][nnd]
```

```
int ielm[4][2517507]; //[4][nel]
int iseg[2][3029220]; //[2][nsg]
int ibel[5][98998]; //[5][nbe]
double density[462216]; //[nnd]
double pressure[462216]; //[nnd]
/**/
//** NASA
double coor[3][381740]; //[3][nnd]
int ielm[4][2124680]; //[4][nel]
int iseg[2][2531602]; //[2][nsg]
int ibel[5][50370]; //[5][nbe]
double density[381740]; //[nnd]
double pressure[381740]; //[nnd]
double Vx[381740], Vy[381740], Vz[381740];//[nnd]
/**/
/** NASA2
double coor[3][122288]; //[3][nnd]
int ielm[4][665109]; //[4][nel]
int iseg[2][801433]; //[2][nsg]
int ibel[5][28074]; //[5][nbe]
double density[122288]; //[nnd]
double pressure[122288]; //[nnd]
double Vx[122288], Vy[122288], Vz[122288];//[nnd]
/**/
/** cavity
double coor[3][210611]; //[3][nnd]
int ielm[4][139542]; //[4][nel]
int iseg[2][629236]; //[2][nsg]
int ibel[5][139542]; //[5][nbe]
double density[210611]; //[nnd]
double pressure[210611]; //[nnd]
double Vx[210611], Vy[210611], Vz[210611];//[nnd]
/**/
/** hypersonic
double coor[3][84330]; //[3][nnd]
int ielm[4][55851]; //[4][nel]
int iseg[2][251884]; //[2][nsg]
int ibel[5][55851]; //[5][nbe]
double density[84330]; //[nnd]
double pressure[84330]; //[nnd]
double Vx[84330], Vy[84330], Vz[84330];//[nnd]
/**/

int lbe[8];
double Machinf;
double gamma = 1.3;
```

```
void ncstatus(int status)
{
   if(status != NC_NOERR){
   fprintf(stderr, "%s\n", nc_strerror(status));
   exit(-1);
   }
}

void readgeom(char *name)
{
         int ncid;
         int nndVar, nelVar, nsgVar;
         int nbeVar, nwlVar, nsdVar, nsfVar;
         int coorVar;
         int ielmVar, isegVar, ibelVar, lbeVar;

         char filename[256];
         strcpy(filename,name);
         strcat(filename, ".nc3d");

         ncstatus(nc_open(filename,0,&ncid));

         ncstatus(nc_inq_dimid(ncid, "nnd", &nndVar));
         ncstatus(nc_inq_dimid(ncid, "nel", &nelVar));
         ncstatus(nc_inq_dimid(ncid, "nsg", &nsgVar));
         ncstatus(nc_inq_dimid(ncid, "nbe", &nbeVar));
         //ncstatus(nc_inq_dimid(ncid, "nwl", &nwlVar));
         //ncstatus(nc_inq_dimid(ncid, "nsd", &nsdVar));
         ncstatus(nc_inq_dimid(ncid, "nsf", &nsfVar));

         ncstatus(nc_inq_dimlen(ncid, nndVar, (size_t *) &nnd));
   ncstatus(nc_inq_dimlen(ncid, nelVar, (size_t *) &nel));
   ncstatus(nc_inq_dimlen(ncid, nsgVar, (size_t *) &nsg));
         ncstatus(nc_inq_dimlen(ncid, nbeVar, (size_t *) &nbe));
   //ncstatus(nc_inq_dimlen(ncid, nwlVar, (size_t *) &nwl));
   //ncstatus(nc_inq_dimlen(ncid, nsdVar, (size_t *) &nsd));
   ncstatus(nc_inq_dimlen(ncid, nsfVar, (size_t *) &nsf));

         ncstatus(nc_inq_varid(ncid, "COOR", &coorVar));
   ncstatus(nc_inq_varid(ncid, "IELM", &ielmVar));
   ncstatus(nc_inq_varid(ncid, "ISEG", &isegVar));
   ncstatus(nc_inq_varid(ncid, "IBEL", &ibelVar));
   ncstatus(nc_inq_varid(ncid, "LBE",  &lbeVar));

         ncstatus(nc_get_var_double(ncid, coorVar, &coor[0][0]));
```

```
          ncstatus(nc_get_var_int(ncid, ielmVar, &ielm[0][0]));
          ncstatus(nc_get_var_int(ncid, isegVar, &iseg[0][0]));
          ncstatus(nc_get_var_int(ncid, ibelVar, &ibel[0][0]));
          ncstatus(nc_get_var_int(ncid, lbeVar,  &lbe[0]));

          ncstatus(nc_close(ncid));

}

void readunk(char *name)
{
          int ncid;
          int rhoVar, preVar, xvelVar, yvelVar, zvelVar;
          int MachVar, gamVar;

          char filename[256];
          strcpy(filename,name);
          strcat(filename, ".unk");

          ncstatus(nc_open(filename,0,&ncid));

          ncstatus(nc_inq_varid(ncid, "Density", &rhoVar));
          ncstatus(nc_inq_varid(ncid, "Pressure", &preVar));
          ncstatus(nc_inq_varid(ncid, "Xvelocity", &xvelVar));
   ncstatus(nc_inq_varid(ncid, "Yvelocity", &yvelVar));
   ncstatus(nc_inq_varid(ncid, "Zvelocity", &zvelVar));
          //ncstatus(nc_inq_attid(ncid, MachVar, "xmi", &MachVar, &Machinf));
          //ncstatus(nc_inq_att(ncid, "gam", &gamVar));

          ncstatus(nc_get_var_double(ncid, rhoVar,  &density[0]));
          ncstatus(nc_get_var_double(ncid, preVar,  &pressure[0]));
          ncstatus(nc_get_var_double(ncid, xvelVar, &Vx[0]));
   ncstatus(nc_get_var_double(ncid, yvelVar, &Vy[0]));
   ncstatus(nc_get_var_double(ncid, zvelVar, &Vz[0]));
          //ncstatus(nc_get_var_double(ncid, MachVar, &Machinf));
          //ncstatus(nc_get_var_double(ncid, gamVar, &gamma));

          ncstatus(nc_close(ncid));
}

int main( int argc, char *argv[] )
{
  //Variables
  int i;
  int j;
  char userV;
```

```
 double pinf;
 static vtkIdType thenode[1]; //nodes
 static vtkIdType npts[4]; //nodal info for elements
 static vtkIdType sol[3]; //solid surfaces
 static vtkIdType temp[1];

 // Read in data file
 char filename[256];
 cout << "Please enter file name " <<endl;
 cin >> filename;

 readgeom(filename); //read in geometry
 readunk(filename); //read in unk

 //output for testing
 cout << "nnd = " << nnd << endl;
 cout << "nel = " << nel << endl;
 cout << "nsg = " << nsg << endl;
 cout << "nbe = " << nbe << endl;

 /***************************************
   Source Data Manipulation
 ***************************************/
 //Setup unstructured grids
 vtkUnstructuredGrid *node = vtkUnstructuredGrid::New (); //make nodes
 node->Allocate (1000); //do first

 vtkUnstructuredGrid *element = vtkUnstructuredGrid::New (); //make grid
 element->Allocate (1000); //do first

 vtkUnstructuredGrid *boundary = vtkUnstructuredGrid::New (); //make solids
surface
 boundary->Allocate (1000); //do first

 //Setup nodes and coords (vertex (1) in vtk)
 vtkPoints *nodes = vtkPoints::New();
 for (j = 0; j < nnd; j++)
 {
         //get points x, y, z
         nodes->InsertPoint(j, coor[0][j], coor[1][j], coor[2][j]);
         //to be able to input actual nodes in render
         thenode[0] = j;
         node->InsertNextCell (1, 1, thenode);
 }

 // Step through elements (tetra (10) in vtk)
```

```
  for (int iel = 0; iel < nel; iel++)
  {
    npts[0] = ielm[0][iel]-1;
    npts[1] = ielm[1][iel]-1;
    npts[2] = ielm[2][iel]-1;
    npts[3] = ielm[3][iel]-1;
            element->InsertNextCell (10, 4, npts);
  }

  // Step through solid surfaces (tri (5) in vtk)
  for (int ibe = 0; ibe < lbe[1]; ibe++)
  {
    sol[0] = ibel[0][ibe]-1;
    sol[1] = ibel[1][ibe]-1;
    sol[2] = ibel[2][ibe]-1;
            boundary->InsertNextCell (5, 3, sol);
  }

  //add symetry planes
  for (ibe = lbe[2]; ibe < lbe[3]; ibe++)
  {
    sol[0] = ibel[0][ibe]-1;
    sol[1] = ibel[1][ibe]-1;
    sol[2] = ibel[2][ibe]-1;
            boundary->InsertNextCell (5, 3, sol);
  }

  node->SetPoints (nodes);
  element->SetPoints (nodes);
  boundary->SetPoints (nodes);

  //For Loop
//for (int oops; oops<100; oops++)
//{

  //Determine which variable to read and write
  cout << "Please enter variable you wish to plot: " <<endl;
  cout << "1 = Pressure " << endl <<"2 = Density" << endl << "3 = Velocity
Magnitude" << endl;
  cout << "4 = Mach #" << endl << "5 = Entropy" << endl;
  cout << "6 = Total Enthalpy" << endl << "7 = Total Energy" << endl << "8
= Internal Energy" << endl;
  cin >> userV;

  //if (userV == '0')
              //break;
```

```
 vtkFloatArray *newdata = vtkFloatArray::New ();

 //pinf = 1/(gamma*Machinf*Machinf); //find pinf
 pinf = 1.0f/(1.3*0.65*0.65); //find pinf

        //Cp
 if (userV =='1')
        for (int inode = 0; inode < nnd; inode++)
        {
                newdata->InsertValue (inode, 2*(pressure[inode]-pinf));
        }
 //density
 if (userV == '2')
        for (int inode = 0; inode < nnd; inode++)
        {
                 newdata->InsertValue (inode, density[inode]);
        }
        //velocity man
 if (userV == '3')
        for (int inode = 0; inode < nnd; inode++)
        {

                 newdata->InsertValue (inode,
sqrt(Vx[inode]*Vx[inode]+Vy[inode]*Vy[inode]+Vz[inode]*Vz[inode]));
        }
        //Mach
 if (userV == '4')
        for (int inode = 0; inode < nnd; inode++)
        {
                 newdata->InsertValue (inode,
sqrt(density[inode]*(Vx[inode]*Vx[inode]+Vy[inode]*Vy[inode]+Vz[inode]*Vz
[inode])/(gamma*pressure[inode])));
        }
        //entropy
 if (userV == '5')
        for (int inode = 0; inode < nnd; inode++)
        {
                 newdata->InsertValue (inode,
pressure[inode]/(pinf*pow(density[inode],gamma))-1.0f);
        }
        //Total Enthalpy
 if (userV == '6')
        for (int inode = 0; inode < nnd; inode++)
        {
```

```
                    newdata->InsertValue (inode, (gamma/(gamma-
1.0f))*pressure[inode]+0.5*density[inode]*(Vx[inode]*Vx[inode]+Vy[inode]*V
y[inode]+Vz[inode]*Vz[inode]));
          }
          //Total energy
  if (userV == '7')
          for (int inode = 0; inode < nnd; inode++)
          {
                    newdata->InsertValue (inode, (1.0f/(gamma-
1.0f))*pressure[inode]+0.5*density[inode]*(Vx[inode]*Vx[inode]+Vy[inode]*V
y[inode]+Vz[inode]*Vz[inode]));
          }
          //internal energy
  if (userV == '8')
          for (int inode = 0; inode < nnd; inode++)
          {
                    newdata->InsertValue (inode, (1.0f/(gamma-
1.0f))*pressure[inode]/density[inode]);
          }

  //input data
  node->GetPointData()->SetScalars (newdata);
  element->GetPointData()->SetScalars (newdata);
  boundary->GetPointData()->SetScalars (newdata);
  nodes->Delete();

/************************************
 Mappers
************************************/
  //node mapper
  vtkDataSetMapper *nodeMapper = vtkDataSetMapper::New();
     nodeMapper->SetInput(node);
     nodeMapper->ImmediateModeRenderingOn();

  //elements
  vtkDataSetMapper *elementMapper = vtkDataSetMapper::New();
     elementMapper->SetInput(element);
     elementMapper->ImmediateModeRenderingOn();

  //solid boundary
  vtkDataSetMapper *boundaryMapper = vtkDataSetMapper::New();
     boundaryMapper->SetInput(boundary);
     boundaryMapper->ImmediateModeRenderingOn();

/************************************
 Actors
```

```
***********************************/
 //nodes
 vtkActor *nodeActor = vtkActor::New();
    nodeActor->SetMapper(nodeMapper);
    nodeActor->GetProperty()->SetColor(0,0,0); //black
          nodeActor->GetProperty()->SetPointSize(2.0);
    nodeActor->AddPosition(0,0.001,0);

 //element
 vtkActor *elementActor = vtkActor::New();
    elementActor->SetMapper(elementMapper);
    //elementActor->GetProperty()->SetColor(.8,.8,.8); //gray
    elementActor->AddPosition(0,0.001,0);

 //solid surface
 vtkActor *boundActor = vtkActor::New();
    boundActor->SetMapper(boundaryMapper);
    //boundActor->GetProperty()->SetColor(0,0,1); //blue
    boundActor->AddPosition(0,0.001,0);
          //boundActor->GetProperty()->SetInterpolationToGouraud();
          //boundActor->GetProperty()->SetRepresentationToWireframe();

 //wire frame from elements
 vtkActor *wireActor = vtkActor::New();
    wireActor->SetMapper(elementMapper);
          wireActor->GetProperty()->SetRepresentationToWireframe();
    wireActor->GetProperty()->SetColor(0,0,0); //black

 // Now create a lookup table that consists of the full hue circle
 vtkLookupTable *hueLut = vtkLookupTable::New ();
          //hueLut->SetTableRange(0,1);
          hueLut->SetTableRange (boundary->GetScalarRange());
          hueLut->SetHueRange (.83,0); // 240/360=blue
          hueLut->SetSaturationRange (1, 1);
          hueLut->SetValueRange (1, 1);
          hueLut->Build ();     //effective built

 //Color Mapping
 vtkDataSetMapper *datamapper = vtkDataSetMapper::New();
          datamapper->SetInput(boundary);
          datamapper->SetScalarRange(boundary->GetScalarRange());
    datamapper->SetLookupTable(hueLut);
          //datamapper->GetProperty()->SetInterpolationToGouraud();
 vtkActor *dataActor = vtkActor::New();
          dataActor->SetMapper(datamapper);
 //Scalar color bar
 vtkScalarBarActor *scalarBar = vtkScalarBarActor::New();
          //scalarBar->SetLookupTable(datamapper->GetLookupTable());
          scalarBar->SetOrientationToVertical();

          if (userV == '1')
          scalarBar->SetTitle("Cp");
          if (userV == '2')
          scalarBar->SetTitle("Density");
          if (userV == '3')
          scalarBar->SetTitle("Velocity Mag");
          if (userV == '4')
          scalarBar->SetTitle("Mach #");
          if (userV == '5')
          scalarBar->SetTitle("Entropy");
          if (userV == '6')
          scalarBar->SetTitle("Total Enthalpy");
          if (userV == '7')
          scalarBar->SetTitle("Total Energy");
          if (userV == '8')
          scalarBar->SetTitle("Internal Energy");

          scalarBar->SetNumberOfLabels(2);
          scalarBar->GetTitleTextProperty()->SetColor(0,0,0); //Title is black
          scalarBar->GetLabelTextProperty()->SetColor(0,0,0); //Label text is
black
          scalarBar->SetLookupTable(hueLut);


/************************************
 Render Window
************************************/
 vtkRenderer *renderer = vtkRenderer::New();
 vtkRenderWindow *renWin = vtkRenderWindow::New();
   renWin->AddRenderer(renderer);
 vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
   iren->SetRenderWindow(renWin);

 //renderer->AddActor(elementActor);
          //renderer->AddActor(nodeActor);
          //renderer->AddActor(boundActor);
          //renderer->AddActor(wireActor);
          renderer->AddActor(scalarBar);
          renderer->AddActor(dataActor);
          renderer->SetBackground(1,1,1);
    renderer->GetActiveCamera()->Elevation(0.0);
```

```
        renderer->GetActiveCamera()->Azimuth(0.0);
        renderer->GetActiveCamera()->Zoom(0);

            renWin->SetSize(400,400);

      cout << "Point Alpha " <<endl;
      // interact with data
      renWin->Render();
      cout << "Point Bravo " <<endl;

      iren->Start();

     //} //End for loop

      // Clean up
      renderer->Delete();
      renWin->Delete();
      //iren->Delete();  //commented to stop error when closing
      nodes->Delete();
      element->Delete();
      elementMapper->Delete();
      elementActor->Delete();
      boundary->Delete();
      boundaryMapper->Delete();
      boundActor->Delete();
      wireActor->Delete();
      newdata->Delete();

      return 0;
}
```

# VITA

Cody Wayne Pinkerman

Candidate for the Degree of

Master of Science

Thesis: ADVANCEMENT OF COMPUTATIONAL FLUID DYNAMICS VISUALIZATION FOR FINITE ELEMENT APPLICATIONS

Major Field: Mechanical and Aerospace Engineering

Biographical:

Personal Data: Born in Lawton, Oklahoma, October 1985 to Charles and Sandra Pinkerman;

Education: Student at Big Pasture School High School, Randlett, Oklahoma till May 2002; Graduated from the Oklahoma School of Science and Mathematics, Oklahoma City, Oklahoma in May 2004; Completed the requirements for the Bachelor of Science in Mechanical and Aerospace Engineering at Oklahoma State University, Stillwater, Oklahoma in May, 2008; Completed the requirements for the Master of Science in Mechanical and Aerospace Engineering at Oklahoma State University, Stillwater, Oklahoma in July, 2010.

Experience: Teaching Assistant, Oklahoma State University School of Mechanical and Aerospace Engineering, 2008-2010; Intern, Frontier Electronic Systems, Aerospace Systems Division, 2008.

Professional Memberships: American Institute of Aeronautics and Astronautics (AIAA).

Name: Cody Wayne Pinkerman Date of Degree: July, 2010

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: ADVANCEMENT OF COMPUTATIONAL FLUID DYNAMICS VISUALIZATION FOR FINITE ELEMENT APPLICATIONS

Pages in Study: 146 Candidate for the Degree of Master of Science

Major Field: Mechanical and Aerospace Engineering

Scope and Method of Study: This thesis focuses on a current CFD visualization program: current capabilities and required upgrades. New features are added to the CFD visualization program to allow for continued operation. Drawbacks of the previous visualization program in OpenGL are also analyzed during this process, and a new visualization program is envisioned. The work on the new visualization system in VTK is shown in detail along with the vision of the final product, which includes later research in a remeshing system for geometry files used in CFD visualization.

Findings and Conclusions: The additions of four new plot-able variables were added to the previous CFD visualization program: velocity magnitude, total energy, internal energy, and vorticity. Along with these new variables a minimum and maximum feature was incorporated which allows for the range of the color and data scale to be changed based on user input.

Programming for new visualization program was started. This thesis focuses on the initial phases of the program: learning new tool kit architecture and programming methods; integrating new tool kit with current geometry and solution file types. A program was written that has the capabilities to plot an object's geometry and eight different property distributions. Multiple test cases were plotted by the new visualization program and the current program to compare and validate the new system. Recommendations were made for the continued progression on the new visualization program.

ADVISER'S APPROVAL: Dr. Andrew S. Arena Jr.