Computational Fluid Dynamics for Propulsion Technology: Final Report

Geometric Grid Visualization in
CFD-Based Propulsion Technology Research

Contract Number: NAS8-36955/DO 85
Report Number: CCFD-92-01

March 17, 1992

Prepared by:

John Ziebarth and Doug Meyer

Department of Computer Science
University of Alabama - Huntsville
Huntsville, AL 35899

Submitted to:

CFD Branch (ED32)
NASA Marshall Space Flight Center
Huntsville, AL 35812

(NASA-CR-184327) COMPUTATIONAL FLUID DYNAMICS FOR PROPULSION TECHNOLOGY: GEOMETRIC GRID VISUALIZATION IN CFD-BASED PROPULSION TECHNOLOGY RESEARCH Final Report No. 13, 20 Dec. 1990 - 1 Apr. 1992 (Alabama G3/34 0086920

N92-23568

Unclas

Primary Project Overview

In the early stages of the contract effort, several meetings were held with members of the computational fluid dynamics branch at NASA's Marshall Space Flight Center. It was the intended purpose of these meetings to discuss what research activities in the area of scientific visualization would be most beneficial in light of both current and long-term activities. The three primary contacts: Kevin Tucker of the Combustion Devices Technology Team, Robert Garcia of the Pump Technology Team, and Lisa Griffin of the Turbine Stage Technology Team, all expressed the opinion that this was a fairly new undertaking within the organization. As such, a cohesive approach for the integration of this type of technology into existing branch systems and procedures had not been formulated. It was mutually agreed that several immediate visualization needs existed from which it would perhaps be possible to extrapolate longer term development goals. To that end, three projects were identified for immediate investigation.

The first project, initially conceived by Robert Garcia, dealt with the development of a software system for performing interactive geometry visualization of the shuttle's main engine fuel pump impeller. Currently under great scrutiny within the branch, the impeller geometry was already defined numerically through its associated computational grids. No software existed, however, which allowed the structure of these grids to be viewed in a photo-realistic fashion. In addition to viewing the geometry of the impeller itself, Garcia noted that eventually it would be necessary to visually interpret data generated from simulations applied against these grids. To that end, he thought consideration should be given to including within the software the capability of overlaying numerical values with the visual depiction of the grid geometry.

The second project, initiated by Kevin Tucker, dealt with the development of a comparable visualization facility for use on the shuttle's film/convective dump cooled nozzle. Here too, the geometry of the nozzle was defined inasmuch as the computational grids had been constructed and were being utilized by several contracting agencies. The capability of visually depicting the structure of the nozzle components in a realistic fashion, however, did not exist. Tucker also expressed that eventually his group would have need of visually interpreting simulation data and that the overlaying of simulation and geometry data in a common graphical format would be ideal for their purposes. A number of independent contractors were in the process of performing simulations utilizing a number of distinct simulation codes. As such, it was considered a necessity to have available just such a standardized display mechanism so that comparisons between the results could be made and trends more easily identified between the distinct data sets.

Lastly, a third project was outlined by Lisa Griffin. In this instance, the development of a software system with capabilities comparable to those discussed by Garcia and Tucker was requested. The system, however, would necessarily require tailoring to permit the visualization of several distinct types of turbines currently under study within her group.

Design Considerations

Although distinct shuttle main engine components were being considered in each of the three projects mentioned, the similarity of the visualization requirements among the three was significant. It was therefore decided that an attempt would be made to develop a common environment which would be robust enough to address the collective geometric visualization needs of the three projects. Although it was considered an attainable goal from the start, several design issues required further investigation prior to development.

Common Data Format

One of the initial problems to be overcome in the development of the proposed visualization environment was one of data management. Given the distinct methods and procedures of the various groups involved, no single, unified format existed for the storage of grid and simulation information. In the long term, this would clearly seem to be an inhibiting factor to the free exchange for data between NASA and its contracting agencies. Attempting to develop a common format for the encapsulation of such data, albeit necessary, proved to be a formidable task.

In the visualization of CFD simulations, two related, but distinct sets of information must exist in order for rendering to take place. The most obvious is perhaps the numeric results from the simulations themselves. These values, whether they be scalar or vector, represent such metrics as temperature, pressure, or velocity and as such implicitly carry both meaning and dimension. In CFD simulations, however, where a bounded region of an n-dimensional space is being investigated, the simulation results by themselves are not sufficient for visualization. What is required is the specification of a spatial reference over which the data is associated; an explicit or implicit enumeration of the location of each data point in the appropriately dimensioned space. Only with this is the specification of the information complete, with the triplet of *value*, *unit*, and *location* specified for each data point.

In most current implementations, the spatial reference is explicitly imbedded within the data, with coordinates usually encoded as separate fields within the same file or database element. Although this is a simple and straightforward strategy, it ignores the fact that the two are actually distinct pieces of information. As a result, several benefits that can be derived from their separate treatment are hidden.

The first benefit of segregating the data from the specification of a spatial reference is in the flexibility of mappings it encourages within the visualization process. In a system supporting this distinction, data can be visualized in whatever coordinate frame or grid geometry best supports the interrogation effort at hand, even though it may be entirely different from that under which it was originally generated. For example, one class of CFD-generated dataset often used in the visualization process represents simulated surface characteristics of various blade assemblies, with the assembly grids being represented as two-dimensional mesh objects defined in three dimensional space. Although the visualization of these values using the actual grid geometry produces a visually realistic image, such as that shown in Figure 1, the physical characteristics of the object can actually hide details from the

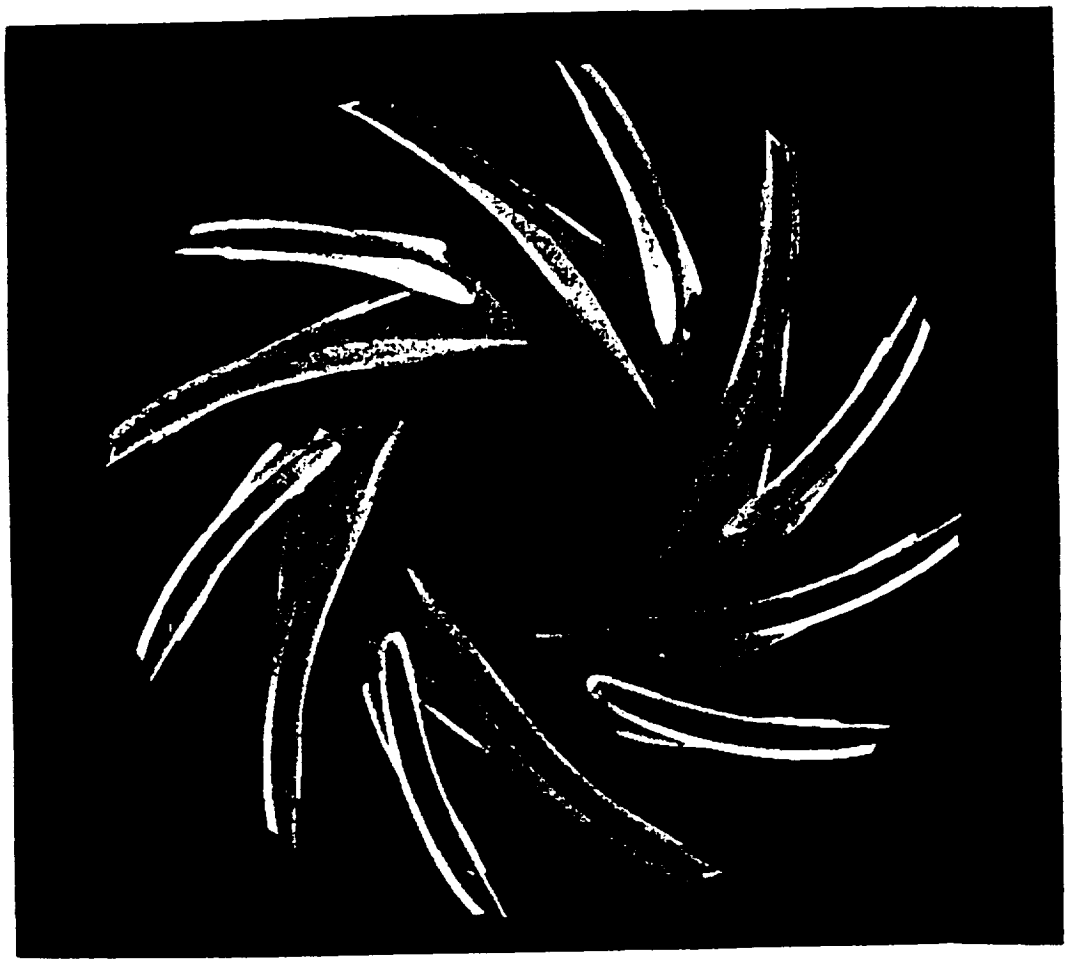


Figure 1. The hiding of information areas due to geometric constraints.

researcher. Such is the case in Figure 1, where the curve of the blades obscure portions of the data. Under situations such as this, it would be beneficial to be able to remap the data onto a flat cartesian grid of equal dimension and size. In a system capable of differentiating between spatial reference and data, such remapping would be a simple procedure.

In order to coordinate the two sets of information, a visualization system such as that described above would need to transparently handle the selective utilization of different coordinate frames as well as the remapping of data between them. It would allow a logical binding to be constructed at the time the visualization takes place between the data and the desired spatial reference. The sole basis of this binding would be a correlation between the number and extent of the data's original grid dimensions and that of the target grid. In the example at hand, since the original surface grids were two dimensional, mapping to a flat, 2D cartesian grid of equal size would be a straightforward. Equally simple would be mappings to any two-dimensional coordinate frame, such as polar or geocentric, or any two-dimensional subset of a higher order space

Another advantage to the scheme suggested above is in the efficient application

of rendering techniques during the visualization process. In the field of computer graphics, difficult problems are often solved by transforming them into simpler problems with either identical or sufficiently similar solutions. The remapping of complex grid geometries to simpler grids of equal dimension for the sake of rendering can be viewed as another application of this approach. Take, for example, a CFD simulation wherein pressure values are calculated for the 3D region between the blades of the assembly shown in Figure 1. The grid for this region, although fundamentally cartesian in nature, would be distorted to conform to the characteristics of the bounding surfaces. Compensating for such distortions would severely impact the computational effort required to render an image of the associated data. This same data, however, remapped onto a regular 3D cartesian grid with sides in alignment with the XYZ axes would be relatively simple to render using any one of a number of existing volume visualization algorithms. Trends and anomalies of interest to the researcher would still be present, although their shape and location would be somewhat distorted due to the remapping. Such distortions may or may not be significant to the effort at hand, but the potential performance improvement of such remapping requires that it at least be offered as a capability of the system.

Finally, in what is purely an implementation concern, the ability to handle spatial references as separate entities promotes the idea of sharing such information internally. In the case of large computational grids, the amount of information required to fully specify the location of each data point can consume large quantities of storage. If properly managed by the visualization system, this information can be stored in a shareable form such that only one copy need be maintained, regardless of how many individual datasets are being visualized with it.

Based on these observations, a storage methodology was adopted for eventual use within the proposed CFD visualization environment wherein spatial reference and simulation data are treated as separate entities. The model for the storage of data will contain the following information:

1.) Dimension of original spatial reference.
2.) Number of data values along each dimension.
3.) Data values, sorted spatially based on dimension.

In addition, spatial references will be specified on the basis of the following information:

1.) Coordinate frame (cartesian, spherical, cylindrical, etc.)
2.) Dimensions (if not implied by the coordinate frame).
3.) Data locations, specified in one of the following forms:
    a.) Minimum/maximum values and data counts for each dimension.
    b.) Explicit enumeration of data locations for each axis.
    c.) Explicit coordinates for each data location.

A more complete description of the data structure developed for spatial references developed during the final phases of this project can be found in Appendix A.

Development Platform

Another issue investigated during the contract period was the possibility of implementing the planned CFD visualization system on a variety of computing platforms. Although most users of the proposed environment would likely be utilizing Silicon Graphics workstations, it was considered good design methodology to at least consider implementation on a wider hardware base for the sake of portability. Three graphical environments were investigated: a Silicon Graphics Personal Iris utilizing the GL graphics interface, a generic UNIX platform using the standard MIT X-Window environment, and a high-end IBM PC compatible with floating point hardware utilizing a direct, custom interface to the graphics adapter. Clearly, significant differences existed between the platforms, including such issues as screen resolution, color capability, and storage (primary and secondary). For the most part, it was thought that a majority of these differences could be compensated for in a graceful manner by the visualization system. One issue, however, defied any type of compensation and quickly lead investigations to cease on all but the Silicon Graphics platform. This issue was polygon rendering speed. Of the set of platforms surveyed, only the Silicon Graphics system provided hardware-based rendering assistance; all others relied solely on software implementations. This lead to rendering speed difference of over three orders of magnitude between the fastest software-based system, rendering at 5-10 polygons a second, and the Silicon Graphics system rendering at 5,000-10,000 polygons a second. Given that some of the objects to be rendered would potentially contain as many as a quarter of a million polygons, the only method of compensating between the systems would be to reduce the accuracy and detail of the rendered image on the slower systems. While this reduction might be feasible for the geometry visualization portion of the project, compensating for the reduction when attempting to map numeric simulation data onto the surfaces would be difficult.

Due to the limited polygon rendering speed, further software development focused solely on the Silicon Graphics family of workstations. By utilizing the SGI GL graphics library, portability was at least obtained throughout the SGI family. Although no additional development took place on the remaining systems, it was considered that at some point in the future another effort be made to gauge their viability in a distributed version of the visualization environment. In such an endeavor, most of the time-consuming rendered would take place remotely and the resulting images downloaded and displayed on the less capable systems. Pursuit of this type of enhancement to the overall system structure, however, was considered best delayed until the actually visualization environment had been developed and had time to mature.

Visualization System Software

Based on the design issue detailed above, development began on a software system which ultimately came to be known as *tview*. Short for "Turbine View", this name reflected the system's initial goal which was to provide visual depictions of turbine geometry. As development proceeded, this name became somewhat of a misnomer since the software ultimately evolved into a form capable of visualizing not

only turbines, but impellers, nozzles, and general surfaces-of-revolution.

As an overview, tview is a software tool for creating photo-realistic, three-dimensional graphics displays of computational grids for turbines, impellers, and nozzles. Not only does the system perform hidden surface removal on the rendered objects, but simulates lighting and shading on them as well. The grids to be displayed could represent individual portions of a much larger assembly on which the user wishes to focus. Conversely, they could be physically separate grids which the user wishes to see simultaneously on a single display. Either way, once loaded into tview, the grids can be displayed and manipulated interactively by the user in real time through the use of the system's mouse and "dial box." Provided is the capability of rotating the object about any of the three axes, as well as the ability of zoom in to and out of the object.

Contained in Appendix B is the complete user's manual for the tview system. This documentation was presented to those branch personnel who assisted in the testing of the system software and has undergone several revisions based on suggestions made. It provides an in-depth discussion of both the capabilities of tview and the procedures for its operation. Also contained in Appendix C is a complete source code listing of the tview system. This software was written in the C programming language and utilizes function calls to the Silicon Graphic GL graphics libraries for all screen and rendering operations.

Conclusions

The tview software has been in use within the branch's three project groups for several months. In that time, it has undergone several revisions based on suggestions of the users and has been ported to several different SGI platforms. In all cases, the software has performed as intended and has played a useful role in the operation of the branch. As was intended from its earliest stages, the development and use of the system has produced a number of ideas for potential enhancements to its capabilities as well as methods for integrating visualization technology more and more into the daily operations of the branch.

Future Work

In the course of performing simulations which utilize the techniques of computational fluid dynamics, the shear volume of information produced poses two formidable problems. The first involves the issue of *data interpretation*. In order to address truly complex issues, CFD simulations must track and produce enormous quantities of data. As the volume of information from such a simulation grows, the probability of detecting the more subtle trends and patterns embedded within is reduced proportionally. This is unfortunate since the primary purpose of such simulations is generally to identify these very trends and patterns.

The second issue to be addressed is one of *data management*. The data supplied to and created by a CFD simulation constitute valuable commodities, with value

potentially being measured using any of several metrics. The value of a given dataset could be calculated in terms of the manpower and computation time required to generate it. Value could also be assessed as a function of the monetary and intellectual significance of the information the dataset contains. Regardless of how it is measured, the fact that the data does indeed have value requires that steps be taken to organize and protect it, while still maintaining the necessary level of accessibility within the data's user community. As a means of addressing these requirements, significant research has been performed in the area of high-volume database design. Encompassing both the hardware and the software aspects of building systems capable of dealing with information sized in the terabyte range, the results of this research could certainly be applied in the solution for this particular situation.

The two problems cited above are in no way independent to one another with respect to the current efforts within the CFD branch. Instead they collectively represent a single problem which is in need of a single, cohesive solution.

In developing a data management facility for use within CFD research, it is a necessity that it be capable of concurrently managing three distinct resource types. First are the CFD codes themselves. Given the requirements for a given project or simulation, the effort involved with determining the proper code to utilize can be significant. A well structured management system should not only have the ability to store and retrieve a broad array of codes, but should also provide access and analysis capabilities in an easy-to-use manner. Codes should be characterized and organized by such well-defined analytical features as equation types, grids, chemistries, applications, schemes, solvers, and turbulence models. Once organized by these criteria, characteristics could be presented to the user in an interactive selection format. This would allow a user without complete knowledge of the characteristics of the various codes to access and utilize the available technology.

The ideal data management facility should also provide for the efficient storage and access of an organization's computational grids, as well as the results of simulations that have been performed against them. Given that considerable effort is expended in the creation of these datasets, both in terms of manpower and compute time, it should be apparent that protective mechanisms for controlling access to them are necessary. Based on the broad base of users requiring access, however, this protection mechanism should be scaled appropriately so as not to impede valid research efforts.

Of foremost concern in a database of the type proposed, where three distinct but related data types are being maintained, is consistency and maintenance of all interdependence relationships. For example, a dataset containing the results of a CFD simulation is of little or no use without the spatial reference supplied by the associated computation grid. Likewise, a set of simulation results combined with the associated computation grid may be of limited utility without knowledge of the specific CFD code used to generate them and any limitations it may possess. It is therefore necessary for the data management system to create and maintain a complex network of bindings between the individual member datasets, creating tangible links which represent the logical relationships between the different classes of information.

Appendix A: Spatial Reference Data Structure Definition

NAME

Spatial reference grid file format

DESCRIPTION

This document details the file format used to externally store spatial reference grids.

FILE LAYOUT (C-STYLE FORMAT):

```
unsigned char  Magic_Number[5];     /* UNIX magic number = "CFDGF"          */
unsigned char  Format_Version[5];   /* Format version number = "01.00"      */
unsigned char  Creator_Name[];      /* Variable-length, null-terminated     */
unsigned char  Creator_Dept[];      /* Variable-length, null-terminated     */
unsigned char  Creator_Org[];       /* Variable-length, null-terminated     */
unsigned char  Creator_Email[];     /* Variable-length, null-terminated     */
unsigned char  Creation_Month;      /* Positive integer value ranged 1-12   */
unsigned char  Creation_Day;        /* Positive integer value ranged 1-31   */
unsigned char  Creation_Year;       /* Positive integer value ranged 0-255  */
unsigned char  Creation_Hour;       /* Positive integer value ranged 0-23   */
unsigned char  Creation_Minute;     /* Positive integer value ranged 0-59   */
unsigned char  Description[];       /* Variable-length, null-terminated     */
unsigned char  Grid_Type;           /* Reg./Axis-explicit/Point-explicit    */
unsigned char  Coord_Frame;         /* Cartesian/Cylindrical/Spherical      */
unsigned short Axis1_Dimension;     /* Positive integer ranged 1-65535      */
unsigned short Axis2_Dimension;     /* Positive integer ranged 1-65535      */
unsigned short Axis3_Dimension;     /* Positive integer ranged 1-65535      */
union Point_Specification           /* One UNION member, based on Grid_Type */
   {struct Regular                  /* Point spec. for regular grid         */
      {float Axis1_MinValue;        /* Min. values for each of the 3 axes   */
       float Axis2_MinValue;
       float Axis3_MinValue;
       float Axis1_MaxValue;        /* Max. values for each of the 3 axes   */
       float Axis2_MaxValue;
       float Axis3_MaxValue;};
    struct Axis-explicit            /* Point spec. for axis-explicit grid   */
      {float Axis1_Coord[];         /* Axis1 coordinates of grid points     */
       float Axis2_Coord[];         /* Axis2 coordinates of grid points     */
       float Axis3_Coord[];};       /* Axis3 coordinates of grid points     */
    struct Point-explicit           /* Point spec. for point-explicit grid  */
      {float Coordinate[][3];};     /* 3D coordinates of grid points        */
   };
```

FIELD NAME	DESCRIPTION
Magic_Number	This sequence of five ASCII characters is used to identify the file as containing a grid specification encoded using the UAH format scheme as outlined in this document.
Format_Version	This sequence of five ASCII characters represent the version number of the format specification used to encode the file's contents. If the grid format were to change in the future, this field could be used to identify all the files which needed updating to the new format.
Creator_Name	This field contains a null-terminated ASCII string representing the name of the individual responsible for the creation of the grid. The field itself is variable in length and may contain control characters such as TAB's and NEWLINE's.
Creator_Dept	This field contains a null-terminated ASCII string representing the name of the department responsible for the creation of the grid. The field itself is variable in length and may contain control characters such as TAB's and NEWLINE's.

dimension (i.e. Axis1_Dimension=1), this value can be ignored by an application.

Axis2_MaxValue This scalar, floating point value (IEEE-754 encoding) represents the maximum coordinate on the grid's first dimension axis. If the grid is topologically planar in this dimension (i.e. Axis2_Dimension=1), this value can be ignored by an application.

Axis3_MaxValue This scalar, floating point value (IEEE-754 encoding) represents the maximum coordinate on the grid's first dimension axis. If the grid is topologically planar in this dimension (i.e. Axis3_Dimension=1), this value can be ignored by an application.

Axis-explicit In an axis-explicit grid, the dimension axes of the grid are topologically parallel to the axes of the coordinate frame in which they are defined. Grid point locations are expressed explicitly for each of the utilized dimension axes.

Axis1_Coord This field contains a list of scalar floating point values (IEEE-754 encoding) representing the locations of grid points along the first dimension axis. The number of values in this list is equal to the value of the Axis1_Dimension field.

Axis2_Coord This field contains a list of scalar floating point values (IEEE-754 encoding) representing the locations of grid points along the second dimension axis. The number of values in this list is equal to the value of the Axis2_Dimension field.

Axis3_Coord This field contains a list of scalar floating point values (IEEE-754 encoding) representing the locations of grid points along the third dimension axis. The number of values in this list is equal to the value of the Axis3_Dimension field.

Point-explicit In a point-explicit grid, the dimension axes of the grid are not restricted to being topologically parallel to the axes of the coordinate frame in which they are defined. Grid point locations are represented explicitly by the specification of a complete 3D coordinate for each.

Coordinate This field contains a list of three-element floating point vectors (IEEE-754 encoding) representing the locations of the grid points in the base coordinate frame as defined by the value of Coord_Frame. The total number of vectors contained in this list is equal to product of the Axis1_Dimension, Axis2_Dimension, and Axis3_Dimension fields. In order to retain the connectivity relationship between these locations, the vectors are stored in Axis1-major order.

NOTES

For both Regular and Axis-explicit grids, a one-to-one relationship exists between the three dimension axes of the grid and the three axes of the coordinate from in which the grid is defined. For each supported coordinate frame, that relationship is as follows:

For Coord_Frame=CARTESIAN_COORD:	Axis1	=	X-Axis
	Axis2	=	Y-Axis
	Axis3	=	Z-Axis
For Coord_Frame=SPHERICAL_COORD:	Axis1	=	Angle from +Z-Axis
	Axis2	=	Angle from +X-Axis to the projection of the radius onto the XY-Plane
	Axis3	=	Radial distance from origin
For Coord_Frame=CYLINDRICAL_COORD:	Axis1	=	Angle from +X-Axis to the projection of the radius onto the XY-Plane
	Axis2	=	Radial distance from Z-Axis
	Axis3	=	Distance from XY-Plane

All references to 3D Cartesian coordinate frames assume a right-handed coordinate system. All angle values are measured in radians and increase in a counter-clockwise direction.

Appendix B: TVIEW User's Manual

TVIEW Version 1.01 User's Manual

Software for CFD Grid Visualization

Table of Contents

Section 1: Documentation Conventions

This section briefly explains the symbol conventions used throughout this documentation. There are three basic symbols used in this document:

[...]	Indicates an optional parameter or value.
<...>	Indicates a necessary parameter or value.
|	Separates several alternative choices.

For instance, the syntax:

`Title:` [*Title of display*]

Indicates that the parameter for the keyword "Title:" is optional. There could be default value, if user does not specify one explicitly.

Section 2: Introduction to *tview*

Tview is a tool for creating realistic, three-dimensional graphics displays of computational grids for turbines, impellers, and nozzles. The grids are defined within the system through the use of configuration and data files, the formats of which are described in detail in Section 4. In addition to data points defining the geometry of the grid, other parameters necessary to the rendering of a realistic image, such as color and surface texture, are also specified via configuration files.

The grids to be displayed with *tview* could represent portions of a much larger assembly on which the user wishes to focus. Conversely, they could be individually separate grids that the user wishes to see simultaneously on a single display. Once loaded, the displayed grids can be manipulated interactively through the use of the dial box (See Section 3.2 "*Using the Dial Box*").

The ability to realistically display and interactively manipulate a wide variety of grid geometries makes *tview* a powerful tool for a number of CFD design and analysis applications.

Section 3: Getting Started

Before invoking *tview*, the user must have available the definition of the grids to be displayed as well as the configuration data indicating how they will be combined and rendered on the display. Information on the structure of the configuration and data files will be provided in Section 4.

Section 3.1: Running *tview*

The syntax for the tview command is:

```
tview [config_file]
```

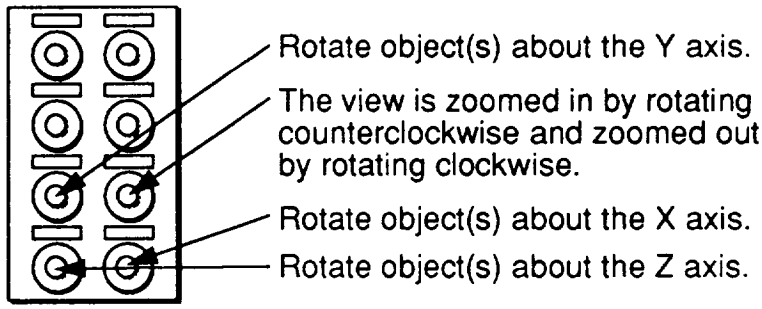
The specification of the configuration file name "*config_file*" is optional. If not specified, the file "input.cfg" in the current directory is used.

Tview parses and reads the contents of the configuration file as well as any data files specified within it. Upon completion of this processing, a standard SGI "window placement" icon is displayed on the screen with the cursor anchored firmly inside it. Through the mouse, the user is able to drag this icon to the screen location where the tview display is to be placed. Once there, by holding down the left mouse button and dragging the cursor across the display, the window can be swept out to the desired size.

Once the tview display window has been established, the system will render an image of the loaded objects. The user is then free to rotate the displayed objects about any of the X, Y, or Z axes, as well as zoom in and out of the object through the use of the system's dial box.

Section 3.2: Using the dial box

The dial box is a peripheral which allows the easy manipulation of the displayed objects without the burden of keyboard intervention. The four bottom dials of the dial box are used by tview for the following purposes:



Section 3.3: Moving and resizing the display window

The display window can be repositioned and resized, using the normal window reposition and resize operations. After such modifications to window take place, however, the displayed object could take a little time to recompute and redraw itself.

Section 3.4: Exiting *tview*

Upon completion of the viewing session, tview can be exited by simply placing the mouse cursor anywhere within the display window and pressing the "ESC" key. This removes the display window from the screen and terminates the tview application.

Section 4: Configuration and Geometry files

The type and content of the visual display is completely defined by the configuration and geometry files provided to tview when it is invoked. The configuration file defines the basic configuration of the objects to be viewed as well as the graphical characteristics of each (color, surface texture, etc.) to be used in the rendering process. The geometry files define the actual geometry of each object to be viewed.

Section 4.1: Configuration file specification

The configuration file is used to define the basic configuration of the display, which includes the objects to be displayed along with their associated parameters. The specific types of information that can be specified in this file are:

1. The title for the display.
2. The axis of rotation for revolute object definitions.
3. The object specification for each of the objects to be displayed. This specification includes the following parameters:
   a. The name of the object to be displayed.
   b. The name of the data file containing the geometry of the object to be displayed.
   c. The color in which for the object is to be displayed.
   d. The type of the object, whether "rotated" or "sor". (surface of revolution)
   e. The specification of rotation parameters for revolute objects, which includes the total angle of rotation, and the number of steps for complete rotation. (Which indirectly specifies the angle of increment to be used for each rotation step)

In order to identify the various parameters within the configuration file, certain *keywords* are used. The details of these keywords, their usage, syntax, and meaning are explained below. It should be noted that the colons (":") specified at the end of most of the keywords are considered part of the keyword and must be used.

Configuration Keyword `Title:`

Syntax:

`Title:` [*User defined title for display*]

Description:

The keyword `Title:` is used to specify the title of the current display. The text string provided immediately after the keyword is placed within the title bar of the display window.

Valid Parameter Values:

A string of characters, to be used as title for the display, not exceeding 79 characters.

Default Parameter Values:

`TurbineView`

Comments and examples:

```
Title: STME Two-Stage Fuel pump impeller.
```

Configuration Keyword `AxisOfRotation:`

Syntax:

`AxisOfRotation: [ X | Y | Z]`

Description:

The keyword `AxisOfRotation:` defines the centerline axis about which revolute objects will be swept. The system supports two types of revolute definitions: "rotated" and "surface-of-revolution". These are explained in greater detail later in this section.

Valid Parameter Values:

`x, y, z, X, Y, Z`

Default Parameter Values:

`Z`

Comments and examples:

```
AxisOfRotation: X
```

Configuration Keyword `Object:`

Syntax:

`Object:` *<User defined name of the object>*

Description:

This keyword is used to mark the beginning of an object definition within the tview environment. Its parameter allows the user to assign a meaningful symbolic name to the associated object. The keywords that follow immediately after are used to specify the remaining attributes of the object. These subordinate keywords are explained below.

ORIGINAL PAGE IS
OF POOR QUALITY

Valid Parameter Values:

A string of characters, defining the user specified name of the object, not more than 79 characters in length.

Default Parameter Values:

None.

Comments and examples:

```
Object: Full Blade - Suction side
```

Object SubKeyword `Geometry:`

Syntax:

`Geometry:` *<Geometry data file name>*

Description:

The `Geometry:` keyword specifies the name of the file containing the geometry data for the object. The format of this file is detailed in Section 4.2.

Valid Parameter Values:

A string of characters, defining a valid file name, not exceeding 255 characters in length.

Default Parameter Values:

None.

Comments and examples:

```
Geometry: ./sufull.dat
```

This specifies that the geometry data for the corresponding object is in the file "sufull.dat" within the current directory.

Object SubKeyword `Type:`

Syntax:

`Type: < Rotated | SOR >`

Description:

The `Type:` keyword specifies the type of the corresponding object.

Valid Parameter Values:

`Rotated, ROTATED, rotated, Sor, SOR, sor`

Default Parameter Values:

None.

Comments and examples:

```
Type: Rotated
```

This specifies that the object is of type rotated.

Object SubKeyword `Color:`

Syntax:

`Color:` *<Red> <Green> <Blue>*

Description:

The `Color:` keyword specifies the color in which the corresponding object is to be displayed. In this situation, color is specified in terms of fractional portions of each of the three additive primary colors: red, green, and blue.

Valid Parameter Values:

Fractions ranging between 0.0 and 1.0, where 0.0 represent no occurrence of the particular primary in the composite color, and 1.0 indicated full intensity of the associated primary.

Default Parameter Values:

Red = 0.4, Green = 0.4, Blue = 0.4

Comments and examples:

```
Color: 1.0 0.0 0.0
```

This would display the corresponding object in a bright red color.

Object SubKeyword `Rotation:`

Syntax:

`Rotation:` *<TotalRotationAngle> <NumberOfSteps>*

Description:

For revolute objects, this keyword specified both the number of times the object is to be replicated about its centerline and the angular region over which these duplications will take place. The `Rotation:` keyword specifies two parameters:

1. The total angle (in degrees) of rotation for the object.
2. The number of steps in which this total angle of rotation is to be achieved. This indirectly defines the angle of increment for each step of rotation.

Valid Parameter Values:

TotalRotationAngle: Any valid signed, floating-point value.
NumberOfSteps: Any positive integer value.

Default Parameter Values:

TotalRotationAngle: 0.0
NumberOfSteps: 1

Comments and examples:

```
Rotation: 360 10
```

This specifies that the object should be rotated 360 degrees in 10 steps, effectively rotating in increments of 36 degrees for each step.

Object SubKeyword `ToggleNormal`

Syntax:

`ToggleNormal`

Description:

The presence of this keyword causes the system to generate surface normals for the associated object which are 180 degrees different from the default. These normals are utilized by the rending system to create a realistic image of the object grid. If the system's default assumption of their direction is incorrect, shading and other visual highlights will also appear incorrect. Reversing the normals in such cases will solve this problem.

Valid Parameter Values:

No parameters required.

Comments and examples:

`ToggleNormal`

Section 4.2: Geometry file specification

The geometry data file (specified by the `Geometry:` keyword described above) contains both the size and geometry information for an object to be displayed. The geometry data is essentially the x, y, and z coordinates of the various grid or contour points. The specification of size, as well as the method by which the grid points will be interconnected in the final mesh, is dictated by the type of object being defined. Currently, two distinct types of objects can be displayed by tview: rotated objects and surfaces of revolution.

Rotated objects:

Geometry files defining rotated objects begin with a specification of the size of the surface mesh. The mesh is assumed to be comprised of a set of "streamlines", which are each defined by an equal number of discrete 3D coordinate points taken along the length. To construct a complete surface mesh, tview automatically connects the coordinate points of adjacent streamlines to form a set of solid, rectangular patches.

The first two values in this type of geometry file are integers indicating the number of coordinate points defining each streamline and the total number of streamlines in the surface. After these values, the program expects to find triplets of floating point values representing the x, y, and z coordinates of the grid points. These triplets are assumed to be organized in streamline order, such that the first N triplets define the first streamline, the second N triplets represent the second streamline, and so on. A sample geometry file for a rotated type object is shown in the following section.

The reasoning behind the name "rotated" for this type of object becomes apparent when the effect of the `Rotation:` keyword is examined. If present, this keyword causes tview to automatically duplicate this single surface definition multiple times about a centerline axis. Visually, each of the duplicates appears to be a separate surface, although internally tview is only storing the geometry for a single instance. This translates into substantial savings in terms of both system memory requirements and overall rendering expense.

Surfaces of revolution:

In several ways, the geometry specification for a surface of revolution is a simplification of that used for the rotated object. In this case, the object is specified by just a single contour or "streamline". To define the complete surface, a continuous sweep of this contour about some centerline axis is required.

The first numeric value contained in a geometry file for this type of object is an integer indicating how many 3D coordinate points are used to specify the surface contour line. After this value, tview expects to find triplets of floating point values representing the x, y, and z coordinates of the contour points.A sample geometry file for a surface of revolution is shown in the following section.

The definition of a surface of revolution requires the use of the `Rotation:` keyword parameter. For these types of objects, the operands of this parameter indicate both the extent and the granularity of the sweep used to define the complete object.

Section 4.3: Sample configuration and geometry files

This section contains examples of valid configuration and geometry files.

Example 1: Rotated object definition

Configuration File: `ex1.cfg`

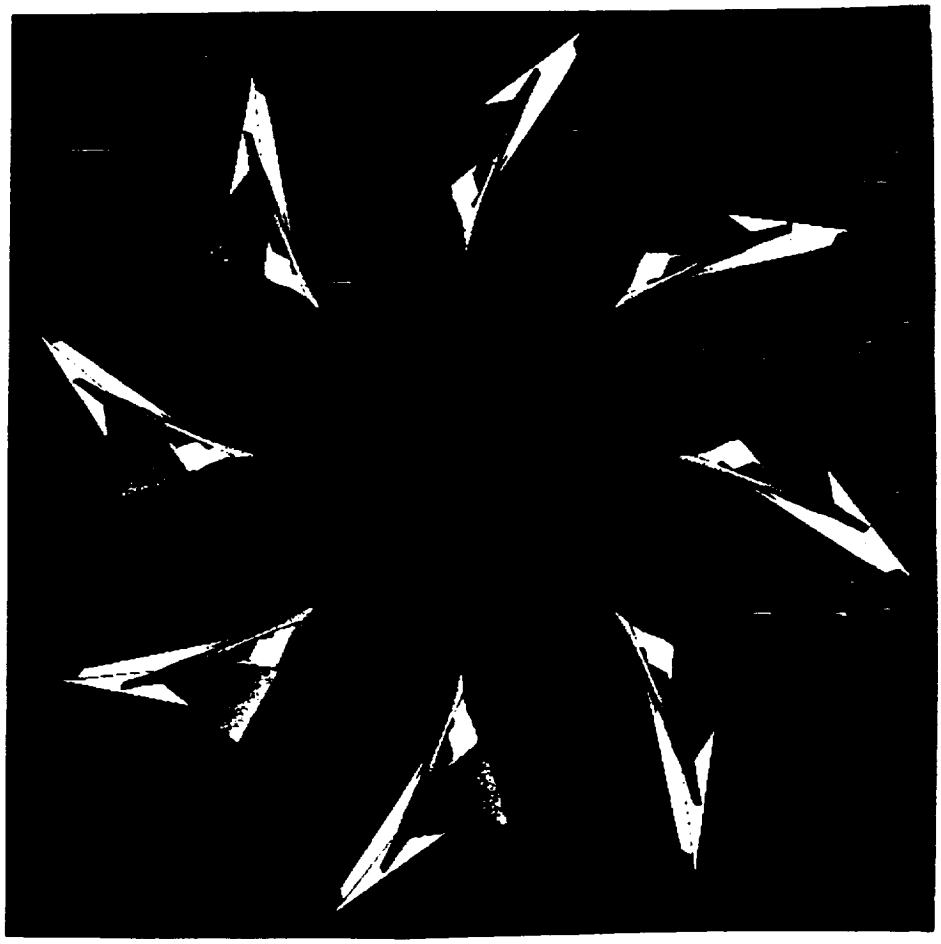
```
Title: Test tview
AxisOfRotation: Z
Object: object 1
   Geometry: object1.dat
   Type: Rotated
   Rotation: 360 8
   Color: 0.4 0.7 0.6
```

Geometry File: `object1.dat`

```
10 2
1.000 -0.0111 2.346
```

```
1.234 -0.0236 2.357
1.245 -0.0321 2.378
1.259 -0.0423 2.421
1.301 -0.0457 2.478
1.324 -0.0534 2.589
1.378 -0.0654 2.603
1.410 -0.0894 2.698
1.489 -0.1984 2.890
1.543 -0.3245 3.587
1.654 -0.4565 3.678
1.756 -0.5432 3.654
1.843  0.4323 3.765
1.932  0.5432 3.876
1.987  0.6548 4.023
1.543  0.8765 4.098
1.675  0.9854 4.124
1.876  1.0987 4.234
1.965  1.0432 4.298
2.098  1.1245 4.324
```

By using the configuration and geometry files above, the following object would be displayed by the command `tview ex1.cfg`



Example 2: Surface of revolution definition

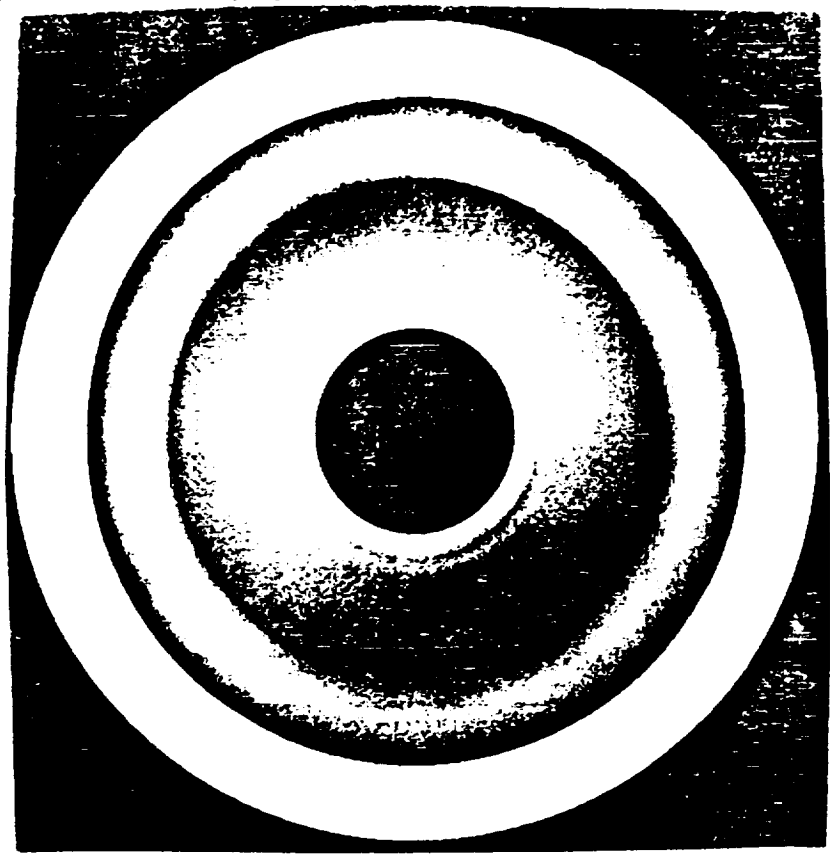
Configuration File: `ex2.cfg`

```
Title: Test tview
AxisOfRotation: Z
Object: object 2
   Geometry: object2.dat
   Type: SOR
   Rotation: 360 90
   Color: 0.4 0.8 0.4
```

Geometry File: object2.dat

```
10
1.000 -0.0111 2.346
1.245 -0.0321 2.378
1.301 -0.0457 2.478
1.378 -0.0654 2.603
1.489 -0.1984 2.890
1.654 -0.4565 3.678
1.843  0.4323 3.765
1.987  0.6548 4.023
1.675  0.9854 4.124
1.965  1.0432 4.298
```

By using the configuration and geometry files above, the following object would be displayed by the command `tview ex2.cfg`



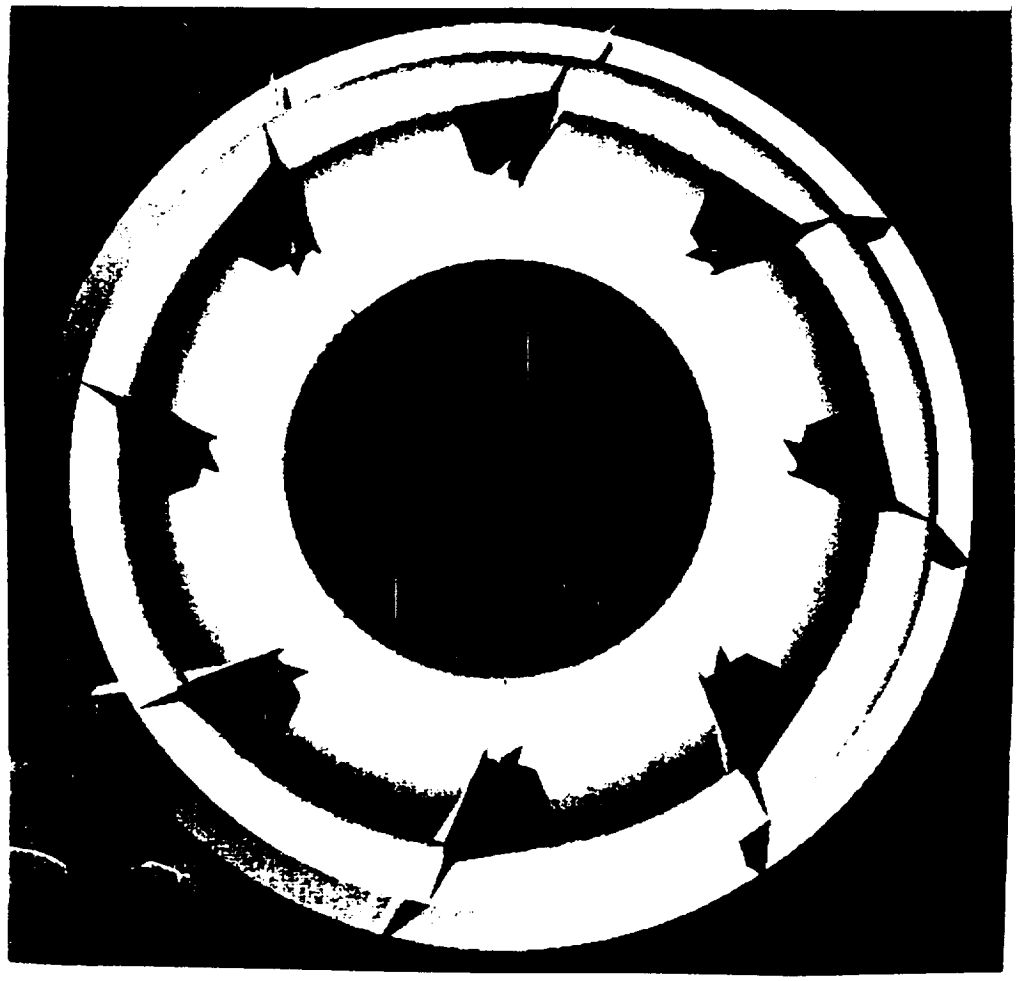
Example 3: Composite object definition

Configuration file: `composite.cfg`

```
Title: Test tview
AxisOfRotation: Z
Object: object 1
   Geometry: object1.dat
   Type: Rotated
   Rotation: 360 8
   Color: 0.4 0.7 0.6
Object: object 2
   Geometry: object2.dat
   Type: SOR
   Rotation: 360 90
   Color: 0.4 0.8 0.4
```

Geometry Files: The geometry files used in this configuration file are those used separately in the preceding two examples: `object1.dat` and `object2.dat`.

By using the files above, the following object would be displayed by the command `tview composite.cfg`



Section 5: Error messages.

The following describes the various error messages issued by tview:

1. **Error encountered during configuration file processing.**
   Indicates that an error was encountered during processing of the configuration file. Additional messages will be printed indicating more closely the nature of the problem.

2. **Error encountered during geometry file processing.**
   Indicates that an was error encountered during processing of the geometry file. Could be due to inability to read the geometry file.

3. **Error encountered during surface normal generation.**
   Indicates that an error was encountered during the automatic generation of surface normals, often due to memory allocation failure.

4. **Unable to access configuration file.**
   Indicates that the configuration file could not be opened for reading.

5. **Line <#>: Invalid axis of rotation specified.**
   Indicates that an invalid axis-of-rotation was specified. The valid values for axis-of-rotation are either x, y, z, X, Y, or Z.

6. **Insufficient memory for dynamic allocation.**
   Indicates that not enough memory was available for the various system data structures used during processing.

7. **Line <#>: Invalid object type specification.**
   Indicates that an invalid object type was specified. Valid object types are either "rotated" or "SOR". The line number indicates where in the configuration file the error occurred.

8. **Line <#>: Invalid color specification.**
   Indicates the line number in the configuration file where an invalid color specification was encountered. Valid color values range from 0.0 to 1.0 for each of the red, green, and blue components.

9. **Line <#>: Invalid rotation specification.**
   Indicates the line number in the configuration file where an invalid rotation specification was encountered. A valid rotation specification consists of one floating point value representing the total rotational angle, and one integer value representing the number of steps taken during traversal of the rotation.

10. **Line <#>: Invalid keyword in configuration file: *<keyword>***
    Indicates that an invalid keyword was specified in the configuration file at the given line number.

11. **No geometry file for object *<objectname>*.**
    Indicates that no geometry file was specified for the object *<objectname>*.

12. **Unable to read geometry file *<filename>* for *<objectname>*.**
    Indicates that the system was unable to open the specified geometry file for reading.

13. **Unable to determine size of geometry file *<filename>*.**
    Indicates that the geometry file did not have the data size specified correctly at the beginning of the file. In the case of type "rotated" objects, the first line of the geometry file should have two integer numbers representing the size of the grid in each dimension. For type "SOR" objects, a single integer representing the number of points on the contour line should be present.

14. **No configuration file name specified.**
    Indicates no name was specified on the command line for the configuration file.

15. **Premature EOF for geometry file *<filename>*.**
    Indicates that during the processing of the geometry file, the number of data entries found was less than the number specified at the beginning of the geometry file.

Appendix C: TVIEW Source Code Listing

TVIEW Source Listing

```
#include <stdio.h>
#include <sys/fcntl.h>
#include <math.h>
#include <gl/gl.h>
#include <gl/device.h>

#define VERSION     "1.01  (10/15/91)"
#define OK              0
#define ERROR          -1
#define MAXLINE        80
#define MAXOBJNAME     80
#define MAXFILENAME   256
#define UNDEFINED       0
#define ROTATED         1
#define SOR             2
#define VIEWDIST       10
#define X               0
#define Y               1
#define Z               2
#define R               0
#define G               1
#define B               2
#define WIREFRAME       0
#define SURFACE         1

struct ObjDef {char   ObjName[MAXOBJNAME];
               char   GeoFile[MAXFILENAME];
               char   Type;
               float  Color[3];
               float  RotAngle;
               float  RotIncr;
               int    RotSteps;
               int    NumPoints;
               int    NumLines;
               int    NormMult;
               float  *Data;
               float  *Norm;
               struct ObjDef *Next;
              };
struct ObjDef *ListHead = NULL;
struct ObjDef *ListTail = NULL;
char   Title[MAXLINE];
int    NoAxes;
char   AxisOfRotation;
static Matrix Identity = {{1.0, 0.0, 0.0, 0.0},
                          {0.0, 1.0, 0.0, 0.0},
                          {0.0, 0.0, 1.0, 0.0},
                          {0.0, 0.0, 0.0, 1.0}};

static float MaterialDef[]  = {DIFFUSE,     0.400,  0.400,  0.400,
                               SPECULAR,    1.000,  1.000,  1.000,
                               AMBIENT,     0.100,  0.100,  0.100,
                               SHININESS,  100.000,
                               LMNULL};

static float LightingDef0[]  = {LCOLOR,      1.000,  1.000,  1.000,
                                POSITION,    2.000,  2.000,  5.000,  0.000,
                                LMNULL};

static float LightingDef1[]  = {LCOLOR,      1.000,  1.000,  1.000,
                                POSITION,   -5.000,  0.000,  5.000,  0.000,
```

```
                                LMNULL};

static float LightingMod[]    = {AMBIENT,      0.400,   0.400,   0.400,
                                 LOCALVIEWER, 1,
                                 LMNULL};

main(argc, argv)
int argc;
char **argv;
{char *config_file = NULL;

 strcpy(Title, "TurbineView");
 NoAxes = FALSE;
 AxisOfRotation = 'z';
 printf("-------------------------------------------------\n");
 printf("         TVIEW Version %s\n", VERSION);
 printf("         Department of Computer Science\n");
 printf("     The University of Alabama in Huntsville\n");
 printf("-------------------------------------------------\n");
 if (argc == 1) config_file = "./input.cfg";
 else config_file = argv[1];

/*-----------------------------------------------------------------------*/
/* Try to get all the object data into memory first. Die on failure.  */
/*-----------------------------------------------------------------------*/
 printf("Object configuration file: %s\n", config_file);
 if (ReadCfgFile(config_file) == ERROR)
   {printf("Error encountered during config file processing.\n");
    FreeObjStructs();
    exit(ERROR);
   }
 if (ReadGeomFiles() == ERROR)
   {printf("Error encountered during geometry file processing.\n");
    FreeObjStructs();
    exit(ERROR);
   }
 if (GenerateNormals() == ERROR)
   {printf("Error encountered during surface normal generation.\n");
    FreeObjStructs();
    exit(ERROR);
   }
 GraphicInit();
 ViewingCycle();
 GraphicTerm();
 FreeObjStructs();
}

/*=======================================================================*/
/* READCFGFILE - Read the configuration file.                            */
/*=======================================================================*/
int ReadCfgFile(CfgFileName)
char *CfgFileName;
{FILE *CfgFile, *fopen();
 char InputLine[MAXLINE], Temp[MAXLINE];
 char *FgetsStat;
 char Field[MAXLINE];
 int  LineCount;

 /*-----------------------------------------------------------------------*/
 /* Make sure we weren't handed a null config file name.                  */
 /*-----------------------------------------------------------------------*/
 if (CfgFileName == NULL)
   {printf("ERROR: No configuration file name specified.\n");
    return(ERROR);
```

```
  }
/*-------------------------------------------------------------------*/
/* Make sure the file actually exists and can be read.               */
/*-------------------------------------------------------------------*/
if ((CfgFile = fopen(CfgFileName, "r")) == NULL)
  {printf("ERROR: Unable to access configuration file.\n");
   return(ERROR);
  }

FgetsStat = fgets(InputLine, MAXLINE, CfgFile);
LineCount = 1;
NormalizeString(InputLine);
sscanf(InputLine, "%s", Field);
/*-------------------------------------------------------------------*/
/* Loop until EOF is encountered in config file.                     */
/*-------------------------------------------------------------------*/
while (FgetsStat != NULL)
     {/*-------------------------------------------------------------*/
      /* Check if current line is a title specification. There should */
      /* only be one, but if there are several it will take the last. */
      /*-------------------------------------------------------------*/
      if (!strcmp(Field, "TITLE:") ||
          !strcmp(Field, "Title:")  ||
          !strcmp(Field, "title:"))
        {strcpy(Title, &(InputLine[strlen(Field)+1]));
         FgetsStat = fgets(InputLine, MAXLINE, CfgFile);
         if (FgetsStat != NULL)
           {LineCount++;
            NormalizeString(InputLine);
            sscanf(InputLine, "%s", Field);
           }
        }
      /*-------------------------------------------------------------*/
      /* Specification of Axis-of-Rotation encountered.               */
      /*-------------------------------------------------------------*/
      else if (!strcmp(Field, "AXISOFROTATION:") ||
          !strcmp(Field, "AxisOfRotation:")  ||
          !strcmp(Field, "axisofrotation:"))
        {sscanf(&(InputLine[strlen(Field)+1]), "%s", Temp);
         if ((Temp[0] == 'x') || (Temp[0] == 'X'))
           AxisOfRotation = 'x';
         else if ((Temp[0] == 'y') || (Temp[0] == 'Y'))
           AxisOfRotation = 'y';
         else if ((Temp[0] == 'z') || (Temp[0] == 'Z'))
           AxisOfRotation = 'z';
         else
           {printf("Line %d: Invalid axis of rotation specified.\n",
                   LineCount);
            return(ERROR);
           }
         FgetsStat = fgets(InputLine, MAXLINE, CfgFile);
         if (FgetsStat != NULL)
           {LineCount++;
            NormalizeString(InputLine);
            sscanf(InputLine, "%s", Field);
           }
        }
      /*-------------------------------------------------------------*/
      /* If they don't want axes, they don't get no stinking axes.    */
      /*-------------------------------------------------------------*/
      else if (!strcmp(Field, "NOAXES") ||
               !strcmp(Field, "NoAxes")  ||
               !strcmp(Field, "noaxes"))
        {NoAxes = TRUE;
```

```
    FgetsStat = fgets(InputLine, MAXLINE, CfgFile);
    if (FgetsStat != NULL)
      {LineCount++;
       NormalizeString(InputLine);
       sscanf(InputLine, "%s", Field);
      }
  }
/*-------------------------------------------------------------*/
/* Check if current line is an object specification.           */
/*-------------------------------------------------------------*/
else if (!strcmp(Field, "OBJECT:") ||
         !strcmp(Field, "Object:") ||
         !strcmp(Field, "object:"))
  {if (ListTail == NULL)
     /*-------------------------------------------------------*/
     /* First object. Allocate block and set head/tail pointers.*/
     /*-------------------------------------------------------*/
     {ListTail = (struct ObjDef *)malloc(sizeof(struct ObjDef));
      if (ListTail == NULL)
        {printf("ERROR: Insufficient memory for dynamic allocation.\n");
         return(ERROR);
        }
      ListHead = ListTail;
     }
   else
     /*-------------------------------------------------------*/
     /* Not first object. Allocate block and tag onto the end.  */
     /*-------------------------------------------------------*/
     {ListTail->Next = (struct ObjDef *)malloc(sizeof(struct ObjDef));
      if (ListTail->Next == NULL)
        {printf("ERROR: Insufficient memory for dynamic allocation.\n");
         return(ERROR);
        }
      ListTail = ListTail->Next;
     }
   /*-------------------------------------------------------*/
   /* Initialize ObjDef structure fields.                   */
   /*-------------------------------------------------------*/
   strcpy(ListTail->ObjName, &(InputLine[strlen(Field)+1]));
   printf("Creating object [%s]\n", ListTail->ObjName);
   ListTail->GeoFile[0] = '\0';
   ListTail->Type       = UNDEFINED;
   ListTail->Color[R]   = 0.4;
   ListTail->Color[G]   = 0.4;
   ListTail->Color[B]   = 0.4;
   ListTail->RotAngle   = 0.0;
   ListTail->RotSteps   = 0;
   ListTail->NumPoints  = 0;
   ListTail->NumLines   = 0;
   ListTail->NormMult   = 1;
   ListTail->Data       = NULL;
   ListTail->Next       = NULL;
   /*-------------------------------------------------------*/
   /* Starting reading lines of associated object information   */
   /* until either EOF or another OBJECT line is encountered.   */
   /*-------------------------------------------------------*/
   FgetsStat = fgets(InputLine, MAXLINE, CfgFile);
   if (FgetsStat != NULL)
     {LineCount++;
      NormalizeString(InputLine);
      sscanf(InputLine, "%s", Field);
     }
   while ((FgetsStat != NULL) &&
          (strcmp(Field, "OBJECT:") &&
```

```
      strcmp(Field, "Object:") &&
      strcmp(Field, "object:")))
{/*-------------------------------------------------------*/
 /* GEOMETRY specification encountered.                   */
 /*-------------------------------------------------------*/
 if (!strcmp(Field, "GEOMETRY:") ||
     !strcmp(Field, "Geometry:") ||
     !strcmp(Field, "geometry:"))
  {sscanf(&(InputLine[strlen(Field)+1]), "%s", ListTail->GeoFile);
   FgetsStat = fgets(InputLine, MAXLINE, CfgFile);
   if (FgetsStat != NULL)
     {LineCount++;
      NormalizeString(InputLine);
      sscanf(InputLine, "%s", Field);
     }
  }
 /*-------------------------------------------------------*/
 /* TYPE specification encountered.                       */
 /*-------------------------------------------------------*/
 else if (!strcmp(Field, "TYPE:") ||
          !strcmp(Field, "Type:") ||
          !strcmp(Field, "type:"))
   {sscanf(&(InputLine[strlen(Field)+1]), "%s", Temp);
    if (!strcmp(Temp, "ROTATED") ||
        !strcmp(Temp, "Rotated") ||
        !strcmp(Temp, "rotated"))
      ListTail->Type = ROTATED;
    else if (!strcmp(Temp, "SOR") ||
             !strcmp(Temp, "sor"))
      ListTail->Type = SOR;
    else
      {printf("Line %d: Invalid object type specification.\n",
              LineCount);
       return(ERROR);
      }
    FgetsStat = fgets(InputLine, MAXLINE, CfgFile);
    if (FgetsStat != NULL)
      {LineCount++;
       NormalizeString(InputLine);
       sscanf(InputLine, "%s", Field);
      }
   }
 /*-------------------------------------------------------*/
 /* COLOR specification encountered.                      */
 /*-------------------------------------------------------*/
 else if (!strcmp(Field, "COLOR:") ||
          !strcmp(Field, "Color:") ||
          !strcmp(Field, "color:"))
   {if (sscanf(&(InputLine[strlen(Field)+1]), "%f%f%f",
        &(ListTail->Color[R]),
        &(ListTail->Color[G]),
        &(ListTail->Color[B])) != 3)
      {printf("Line %d: Invalid color specification.\n",
              LineCount);
       return(ERROR);
      }
    FgetsStat = fgets(InputLine, MAXLINE, CfgFile);
    if (FgetsStat != NULL)
      {LineCount++;
       NormalizeString(InputLine);
       sscanf(InputLine, "%s", Field);
      }
   }
 /*-------------------------------------------------------*/
```

```
            /* ROTATION specification encountered.                    */
            /*-------------------------------------------------------*/
            else if (!strcmp(Field, "ROTATION:") ||
                     !strcmp(Field, "Rotation:") ||
                     !strcmp(Field, "rotation:"))
              {if (sscanf(&(InputLine[strlen(Field)+1]), "%f%d",
                   &(ListTail->RotAngle), &(ListTail->RotSteps)) != 2)
                 {printf("Line %d: Invalid rotation specification.\n",
                         LineCount);
                  return(ERROR);
                 }
               ListTail->RotIncr = ListTail->RotAngle / ListTail->RotSteps;
               FgetsStat = fgets(InputLine, MAXLINE, CfgFile);
               if (FgetsStat != NULL)
                 {LineCount++;
                  NormalizeString(InputLine);
                  sscanf(InputLine, "%s", Field);
                 }
              }
            /*-------------------------------------------------------*/
            /* NORMTOGGLE switch encountered.                        */
            /*-------------------------------------------------------*/
            else if (!strcmp(Field, "TOGGLENORMAL") ||
                     !strcmp(Field, "ToggleNormal") ||
                     !strcmp(Field, "togglenormal"))
              {ListTail->NormMult = -1;
               FgetsStat = fgets(InputLine, MAXLINE, CfgFile);
               if (FgetsStat != NULL)
                 {LineCount++;
                  NormalizeString(InputLine);
                  sscanf(InputLine, "%s", Field);
                 }
              }
            /*-------------------------------------------------------*/
            /* Unknown OBJECT subfield specification encountered.    */
            /*-------------------------------------------------------*/
            else
             {printf("Line %d: Invalid keyword in configuration file. [%s]\n",
                      LineCount, Field);
              return(ERROR);
             }
           }
        }
      /*-----------------------------------------------------------*/
      /* Unknown keyword encountered.                              */
      /*-----------------------------------------------------------*/
      else
        {printf("Line %d: Invalid keyword in configuration file. [%s]\n",
                 LineCount, Field);
         return(ERROR);
        }
    }
 fclose(CfgFile);
 return(OK);
}


/*=============================================================*/
/* NORMALIZESTRING - Remove leading and trailing blanks from a string.*/
/*                   Also gets rid of pesky newlines at the end, too. */
/*=============================================================*/
int NormalizeString(NormString)
 char NormString[];
{int  i, First, Last, Len;
```

```
 First = 0;
 while (NormString[First] == ' ') First++;
 Last = strlen(NormString)-1;
 while ((NormString[Last] == ' ') || (NormString[Last] == '\n')) Last--;
 Len = Last-First+1;
 strncpy(NormString, &(NormString[First]), Len);
 NormString[Len] = '\0';
 return(OK);
}

/*=====================================================================*/
/* FREEOBJSTRUCTS - This routine frees all the dynamically-allocated  */
/*                  structures used to hold the object definitions.   */
/*=====================================================================*/
int FreeObjStructs()
{struct ObjDef *NextObj;
 while (ListHead != NULL)
   {NextObj = ListHead->Next;
    if (ListHead->Data != NULL)
      free(ListHead->Data);
    if (ListHead->Norm != NULL)
      free(ListHead->Norm);
    free(ListHead);
    ListHead = NextObj;
   }
 ListTail = NULL;
 return(OK);
}

/*=====================================================================*/
/* READGEOMFILES - Scan list of objects created by ReadCfgFile and    */
/*                 read in the associated geometry file for each.     */
/*=====================================================================*/
int ReadGeomFiles()
{struct ObjDef *CurrentObj;
 FILE *GeometryFile;
 int  MemSize, i, j, k, index, axis, span;
 double SinVal, CosVal;
 float  Min = 1000000.0, Max = -1000000.0, Mid;

 CurrentObj = ListHead;
 /*-------------------------------------------------------------------*/
 /* Scan through the entire list of objects and load in geometry.     */
 /*-------------------------------------------------------------------*/
 while (CurrentObj != NULL)
   /*-----------------------------------------------------------------*/
   /* Make sure that a geometry file has been defined and can be read.*/
   /*-----------------------------------------------------------------*/
   {if (CurrentObj->GeoFile[0] == '\0')
      {printf("ERROR: No geometry file for object [%s].\n",
              CurrentObj->ObjName);
       return(ERROR);
      }
    if ((GeometryFile = fopen(CurrentObj->GeoFile, "r")) == NULL)
      {printf("ERROR: Unable to read geometry file [%s] for object [%s].\n",
              CurrentObj->GeoFile, CurrentObj->ObjName);
       return(ERROR);
      }
    /*-----------------------------------------------------------------*/
    /* Object is of type ROTATED. Allocate storage and read it in.     */
    /*-----------------------------------------------------------------*/
    if (CurrentObj->Type == ROTATED)
      {if (fscanf(GeometryFile, "%d%d",
              &(CurrentObj->NumPoints), &(CurrentObj->NumLines)) != 2)
```

```
      {printf("ERROR: Unable to determine size of geometry file [%s].\n",
              CurrentObj->GeoFile);
       return(ERROR);
      }
    MemSize = (CurrentObj->NumPoints*CurrentObj->NumLines)*sizeof(float)*3;
    if ((CurrentObj->Data = (float *)malloc(MemSize)) == NULL)
      {printf("ERROR: Insufficient memory for dynamic allocation.\n");
       return(ERROR);
      }
    index = 0;
    for (i=0; i<CurrentObj->NumLines; i++)
       for (j=0; j<CurrentObj->NumPoints; j++)
          {if (fscanf(GeometryFile, "%f%f%f",
                      &(CurrentObj->Data[index]),
                      &(CurrentObj->Data[index+1]),
                      &(CurrentObj->Data[index+2])) != 3)
             {printf("ERROR: Premature EOF for geometry file [%s].\n",
                     CurrentObj->GeoFile);
              return(ERROR);
             }
           /*-------------------------------------------------------*/
           /* Figure Min and Max on axis of rotation so we can      */
           /* center about origin.                                  */
           /*-------------------------------------------------------*/
           if (AxisOfRotation == 'x') axis = index;
           else if (AxisOfRotation == 'y') axis = index+1;
           else axis = index+2;
           if (CurrentObj->Data[axis] < Min)
              Min = CurrentObj->Data[axis];
           if (CurrentObj->Data[axis] > Max)
              Max = CurrentObj->Data[axis];
           index += 3;
          }
  }
/*-------------------------------------------------------------------*/
/* Object is of type SOR (Surface-of-Revolution). The structure      */
/* allocated is actually twice as big as need be because we will     */
/* convert it into patches. We do this by rotating the line one      */
/* one step and creating another line.                               */
/*-------------------------------------------------------------------*/
else if (CurrentObj->Type == SOR)
  {fscanf(GeometryFile, "%d", &(CurrentObj->NumPoints));
   CurrentObj->NumLines = 2;
   MemSize = (CurrentObj->NumPoints)*2*sizeof(float)*3;
   if ((CurrentObj->Data = (float *)malloc(MemSize)) == NULL)
     {printf("ERROR: Insufficient memory for dynamic allocation.\n");
      return(ERROR);
     }
   index = 0;
   /*-------------------------------------------------------------*/
   /* Read the contents of the geometry file first.               */
   /*-------------------------------------------------------------*/
   for (i=0; i<CurrentObj->NumPoints; i++)
      {if (fscanf(GeometryFile, "%f%f%f",
                  &(CurrentObj->Data[index]),
                  &(CurrentObj->Data[index+1]),
                  &(CurrentObj->Data[index+2])) != 3)
         {printf("ERROR: Premature EOF for geometry file [%s].\n",
                 CurrentObj->GeoFile);
          return(ERROR);
         }
       /*-----------------------------------------------------------*/
       /* Figure Min and Max on axis of rotation so we can center */
       /* about origin.                                             */
```

```
            /*-------------------------------------------------------------*/
            if (AxisOfRotation == 'x') axis = index;
            else if (AxisOfRotation == 'y') axis = index+1;
            else axis = index+2;
            if (CurrentObj->Data[axis] < Min)
               Min = CurrentObj->Data[axis];
            if (CurrentObj->Data[axis] > Max)
               Max = CurrentObj->Data[axis];
            index += 3;
           }
         /*-------------------------------------------------------------*/
         /* Now rotate it one step and store result as a second line.   */
         /*-------------------------------------------------------------*/
         CosVal = cos((double)((CurrentObj->RotIncr/360.0)*2*M_PI));
         SinVal = sin((double)((CurrentObj->RotIncr/360.0)*2*M_PI));
         span = CurrentObj->NumPoints * 3;
         for (i=0; i<CurrentObj->NumPoints; i++)
            {if (AxisOfRotation == 'x')
               {CurrentObj->Data[index] =
                     CurrentObj->Data[index-span];
                CurrentObj->Data[index+1] =
                     CurrentObj->Data[index-span+1] * CosVal -
                     CurrentObj->Data[index-span+2] * SinVal;
                CurrentObj->Data[index+2] =
                     CurrentObj->Data[index-span+1] * SinVal +
                     CurrentObj->Data[index-span+2] * CosVal;
               }
             else if (AxisOfRotation == 'y')
               {CurrentObj->Data[index] =
                     CurrentObj->Data[index-span] * CosVal +
                     CurrentObj->Data[index-span+2] * SinVal;
                CurrentObj->Data[index+1] =
                     CurrentObj->Data[index-span+1];
                CurrentObj->Data[index+2] =
                     CurrentObj->Data[index-span+2] * CosVal -
                     CurrentObj->Data[index-span] * SinVal;
               }
             else if (AxisOfRotation == 'z')
               {CurrentObj->Data[index] =
                     CurrentObj->Data[index-span] * CosVal -
                     CurrentObj->Data[index-span+1] * SinVal;
                CurrentObj->Data[index+1] =
                     CurrentObj->Data[index-span] * SinVal +
                     CurrentObj->Data[index-span+1] * CosVal;
                CurrentObj->Data[index+2] =
                     CurrentObj->Data[index-span+2];
               }
             index+=3;
            }
      }
      /*-------------------------------------------------------------*/
      /* Unknown or undefined object type encountered in list.       */
      /*-------------------------------------------------------------*/
      else
        {printf("ERROR: Invalid type for object [%s]\n", CurrentObj->ObjName);
         return(-1);
        }
      CurrentObj = CurrentObj->Next;
     }
   /*-------------------------------------------------------------*/
   /* Scan through object again, this time adjusting value on the axis  */
   /* of rotation to center the object.                           */
   /*-------------------------------------------------------------*/
   Mid = (Min - Max)/2;
```

```
 CurrentObj = ListHead;
 while (CurrentObj != NULL)
   {index = 0;
    for (i=0; i<CurrentObj->NumLines; i++)
       for (j=0; j<CurrentObj->NumPoints; j++)
          {if (AxisOfRotation == 'x') axis = index;
           else if (AxisOfRotation == 'y') axis = index+1;
           else axis = index+2;
           CurrentObj->Data[axis] -= Mid;
           index += 3;
          }
    CurrentObj = CurrentObj->Next;
   }
 return(0);
}

/*--------------------------------------------------------------------*/
/* GENERATENORMALS - Generate surface normals for all objects in list */
/*                   and store them within the object structure.      */
/*--------------------------------------------------------------------*/
int GenerateNormals()
{struct ObjDef *CurrentObj;
 int  span, i, j, index;
 float V0[3], V1[3], mag;
 CurrentObj = ListHead;
 while (CurrentObj != NULL)
   /*--------------------------------------------------------------*/
   /* Dynamically allocate a structure to hold the normals.        */
   /*--------------------------------------------------------------*/
   {CurrentObj->Norm = (float *)malloc((CurrentObj->NumLines-1)
                     * (CurrentObj->NumPoints) * sizeof(float) * 3);
    if (CurrentObj->Norm == NULL)
      {printf("ERROR: Insufficient memory for dynamic allocation.\n");
       return(ERROR);
      }
    span = CurrentObj->NumPoints*3;
    index = 0;
    for (i=0; i<CurrentObj->NumLines-1; i++)
       {for (j=0; j<CurrentObj->NumPoints-1; j++)
           /*-------------------------------------------------------*/
           /* This looks hairy, but its just a cross product between */
           /* the bottom and left edges of each object patch.       */
           /*-------------------------------------------------------*/
           {V0[X] = CurrentObj->Data[index+span]   - CurrentObj->Data[index];
           V0[Y] = CurrentObj->Data[index+span+1] - CurrentObj->Data[index+1];
           V0[Z] = CurrentObj->Data[index+span+2] - CurrentObj->Data[index+2];
            V1[X] = CurrentObj->Data[index+3] - CurrentObj->Data[index];
            V1[Y] = CurrentObj->Data[index+4] - CurrentObj->Data[index+1];
            V1[Z] = CurrentObj->Data[index+5] - CurrentObj->Data[index+2];
            CurrentObj->Norm[index]   = (V0[Y]*V1[Z]) - (V1[Y]*V0[Z]);
            CurrentObj->Norm[index+1] = (V1[X]*V0[Z]) - (V0[X]*V1[Z]);
            CurrentObj->Norm[index+2] = (V0[X]*V1[Y]) - (V1[X]*V0[Y]);
           /*-------------------------------------------------------*/
           /* Make the normal unit length; a requirement of GL.     */
           /*-------------------------------------------------------*/
            mag = sqrt(pow((double)CurrentObj->Norm[index],   (double)2.0)  +
                       pow((double)CurrentObj->Norm[index+1], (double)2.0)  +
                       pow((double)CurrentObj->Norm[index+2], (double)2.0)) *
                       CurrentObj->NormMult;
            CurrentObj->Norm[index]   /= mag;
            CurrentObj->Norm[index+1] /= mag;
            CurrentObj->Norm[index+2] /= mag;
            index += 3;
           }
```

```
            /*-------------------------------------------------------------*/
            /* For the vertices on the end of a strip, we use the normal   */
            /* of the preceding patch since we can't calculate one.        */
            /*-------------------------------------------------------------*/
            CurrentObj->Norm[index]   = CurrentObj->Norm[index-3];
            CurrentObj->Norm[index+1] = CurrentObj->Norm[index-2];
            CurrentObj->Norm[index+2] = CurrentObj->Norm[index-1];
            index += 3;
           }
        CurrentObj = CurrentObj->Next;
       }
}

/*-------------------------------------------------------------------*/
/* GRAPHICINIT - Perform graphic system initialization.              */
/*-------------------------------------------------------------------*/
int GraphicInit()
{keepaspect(3, 2);
 winopen("");
 wintitle(Title);
 doublebuffer();
 RGBmode();
 zbuffer(TRUE);
 backface(TRUE);
 gconfig();
 mmode(MVIEWING);
 /* perspective(900, 1.5, 0.1, 500.0); */
 perspective(400, 1.5, 0.1, 500.0);
 setvaluator(DIAL0, (short)0, (short)-1, (short)360);
 setvaluator(DIAL1, (short)0, (short)-1, (short)360);
 setvaluator(DIAL2, (short)0, (short)-1, (short)360);
 setvaluator(DIAL3, (short)VIEWDIST*10, (short)0,  (short)10000);
 qdevice(DIAL0);
 qdevice(DIAL1);
 qdevice(DIAL2);
 qdevice(DIAL3);
 qdevice(ESCKEY);
 lmdef(DEFMATERIAL, 1, 0, MaterialDef);
 lmdef(DEFLIGHT,    1, 0, LightingDef0);
 lmdef(DEFLIGHT,    2, 0, LightingDef1);
 lmdef(DEFLMODEL,   1, 0, LightingMod);
 lmbind(MATERIAL, 1);
 lmbind(LIGHT0,   1);
 lmbind(LIGHT1,   2);
 lmbind(LMODEL,   1);
 czclear(0x000000, 0x7fffff);
 swapbuffers();
 czclear(0x000000, 0x7fffff);
 swapbuffers();
 qenter(DIAL3, (short)VIEWDIST*10);
 return(0);
}

/*-------------------------------------------------------------------*/
/* GRAPHICTERM - Perform graphic system termination.                 */
/*-------------------------------------------------------------------*/
int GraphicTerm()
{gexit();
}

/*-------------------------------------------------------------------*/
/* VIEWINGCYCLE - Query input from user, render new image. Loop.     */
/*-------------------------------------------------------------------*/
int ViewingCycle()
```

```
{int i;
 short IOdata;
 Device IOdev;
 static float zd = VIEWDIST;
 float  rx=0.0, ry=0.0, rz=0.0,
        rxinc,  ryinc,  rzinc;
 static Matrix Current = {{1.0, 0.0, 0.0, 0.0},
                          {0.0, 1.0, 0.0, 0.0},
                          {0.0, 0.0, 1.0, 0.0},
                          {0.0, 0.0, 0.0, 1.0}};
 static float V0[] = {0.0, 0.0, 0.0},
              Vx[] = {1.0, 0.0, 0.0},
              Vy[] = {0.0, 1.0, 0.0},
              Vz[] = {0.0, 0.0, 1.0};

 /*--------------------------------------------------------------------*/
 /* Loop until the user hits ESC.                                      */
 /*--------------------------------------------------------------------*/
 while ((IOdev=qread(&IOdata)) != ESCKEY)
   {pushmatrix();
    if (IOdev != DIAL3)
      {if (IOdata ==  -1) setvaluator(IOdev, (short)359,(short)-1, (short)360);
       if (IOdata == 360) setvaluator(IOdev, (short)0,  (short)-1, (short)360);
      }
    /*------------------------------------------------------------------*/
    /* Z axis rotation requested.                                       */
    /*------------------------------------------------------------------*/
    if (IOdev == DIAL0)
      {rzinc = rz - ((float)IOdata*10);
       UpdateCurrent('z', rzinc, Current);
       rz -= rzinc;}
    /*------------------------------------------------------------------*/
    /* X axis rotation requested.                                       */
    /*------------------------------------------------------------------*/
    else if (IOdev == DIAL1)
      {rxinc = rx - ((float)IOdata*10);
       UpdateCurrent('x', rxinc, Current);
       rx -= rxinc;}
    /*------------------------------------------------------------------*/
    /* Y axis rotation requested.                                       */
    /*------------------------------------------------------------------*/
    else if (IOdev == DIAL2)
      {ryinc = ry - ((float)IOdata*10);
       UpdateCurrent('y', ryinc, Current);
       ry -= ryinc;}
    /*------------------------------------------------------------------*/
    /* Zoom in/out requested.                                           */
    /*------------------------------------------------------------------*/
    else if (IOdev == DIAL3)
       zd = ((float)IOdata/10);
    else if (IOdev == REDRAW)
      {reshapeviewport();
       czclear(0x000000, 0x7fffff);
       swapbuffers();
       czclear(0x000000, 0x7fffff);
       swapbuffers();
      }
    lookat(0.0, 0.0, zd, 0.0, 0.0, 0.0, 0);
    multmatrix(Current);
    czclear(0x000000, 0x7fffff);
    /*------------------------------------------------------------------*/
    /* Redraw objects. Use wireframe mode if more events are in the     */
    /* queue, or hidden-surface mode if there are not.                  */
    /*------------------------------------------------------------------*/
```

```
    if (qtest() == 0)
      {sleep(2);
       if (qtest() == 0)
         DrawObjList(SURFACE);
       else DrawObjList(WIREFRAME);
      }
    else DrawObjList(WIREFRAME);
    popmatrix();
    /*-------------------------------------------------------------------*/
    /* Draw a set of tilting axes in the lower right of the screen.      */
    /*-------------------------------------------------------------------*/
    if (NoAxes == FALSE)
      {lmbind(LMODEL, 0);
       pushmatrix();
       ortho(-6, 6, -4, 4, 0.1, 500);
       translate(4.5, -2.5, -5);
       multmatrix(Current);
       cpack(0xffff00);
       bgnline(); v3f(V0); v3f(Vx); endline();
       bgnline(); v3f(V0); v3f(Vy); endline();
       bgnline(); v3f(V0); v3f(Vz); endline();
       cpack(0x00ffff);
       cmov(1.2, 0.0, 0.0); charstr("X");
       cmov(0.0, 1.2, 0.0); charstr("Y");
       cmov(0.0, 0.0, 1.2); charstr("Z");
       lmbind(LMODEL, 1);
       popmatrix();
      /* perspective(900, 1.5, 0.1, 500.0); */
       perspective(400, 1.5, 0.1, 500.0);
      }
    swapbuffers();
   }
 return(0);
}

/*-------------------------------------------------------------------*/
/* UPDATECURRENT - Postmultiply the current ModelView matrix by a    */
/*                 single axis rotation matrix.                      */
/*-------------------------------------------------------------------*/
UpdateCurrent(Axis, Angle, CMatrix)
 char   Axis;
 float  Angle;
 Matrix CMatrix;
{pushmatrix();
 loadmatrix(Identity);
 rotate(Angle, Axis);
 multmatrix(CMatrix);
 getmatrix(CMatrix);
 popmatrix();
}

/*-------------------------------------------------------------------*/
/* DRAWOBJLIST - Scan the list of object definitions and draw each   */
/*               in either wireframe or surface mode.                */
/*-------------------------------------------------------------------*/
DrawObjList(DrawType)
 int DrawType;
{struct ObjDef *CurrentObj;
 int i, j, k, index, span;
 /*-------------------------------------------------------------------*/
 /* Render object with lighting model and hidden surface elimination. */
 /* Because the polygons are being treated as a 3D object, each must  */
 /* be rendered twice (once facing each way) to get correct lighting. */
 /*-------------------------------------------------------------------*/
```

```
if (DrawType == SURFACE)
  {lmbind(LMODEL, 1);
   CurrentObj = ListHead;
   while (CurrentObj != NULL)
     {MaterialDef[1] = CurrentObj->Color[R];
      MaterialDef[2] = CurrentObj->Color[G];
      MaterialDef[3] = CurrentObj->Color[B];
      lmdef(DEFMATERIAL, 1, 0, MaterialDef);
      span = CurrentObj->NumPoints * 3;
      for (i=0; i<CurrentObj->RotSteps; i++)
         /*-------------------------------------------------------------*/
         /* Draw the surface facing one way.                            */
         /*-------------------------------------------------------------*/
         {pushmatrix();
          rotate((long)(CurrentObj->RotIncr*i*10), AxisOfRotation);
          index = 0;
          for (j=0; j<CurrentObj->NumLines-1; j++)
             {bgntmesh();
              for (k=0; k<CurrentObj->NumPoints; k++)
                 {n3f(&(CurrentObj->Norm[index]));
                  v3f(&(CurrentObj->Data[index]));
                  v3f(&(CurrentObj->Data[index+span]));
                  index += 3;
                 }
              endtmesh();
             }
          popmatrix();
         }
      /*-------------------------------------------------------------*/
      /* Toggle the surface normals to face the other way.           */
      /*-------------------------------------------------------------*/
      index = 0;
      for (j=0; j<CurrentObj->NumLines-1; j++)
         for (k=0; k<CurrentObj->NumPoints; k++)
            {CurrentObj->Norm[index]   = -CurrentObj->Norm[index];
             CurrentObj->Norm[index+1] = -CurrentObj->Norm[index+1];
             CurrentObj->Norm[index+2] = -CurrentObj->Norm[index+2];
             index += 3;
            }
      for (i=0; i<CurrentObj->RotSteps; i++)
         /*-------------------------------------------------------------*/
         /* Draw the surface facing the other way.                      */
         /*-------------------------------------------------------------*/
         {pushmatrix();
          rotate((long)(CurrentObj->RotIncr*i*10), AxisOfRotation);
          index = ((CurrentObj->NumLines-1)*(CurrentObj->NumPoints)*3)-3;
          for (j=0; j<CurrentObj->NumLines-1; j++)
             {bgntmesh();
              for (k=0; k<CurrentObj->NumPoints; k++)
                 {n3f(&(CurrentObj->Norm[index]));
                  v3f(&(CurrentObj->Data[index]));
                  v3f(&(CurrentObj->Data[index+span]));
                  index -= 3;
                 }
              endtmesh();
             }
          popmatrix();
         }
      /*-------------------------------------------------------------*/
      /* Toggle the surface normals to to the way they began.        */
      /*-------------------------------------------------------------*/
      index = 0;
      for (j=0; j<CurrentObj->NumLines-1; j++)
         for (k=0; k<CurrentObj->NumPoints; k++)
```

```
            {CurrentObj->Norm[index]   = -CurrentObj->Norm[index];
             CurrentObj->Norm[index+1] = -CurrentObj->Norm[index+1];
             CurrentObj->Norm[index+2] = -CurrentObj->Norm[index+2];
             index += 3;
            }
       CurrentObj = CurrentObj->Next;
      }
  }
/*-------------------------------------------------------------------------*/
/* Render object in a wireframe mode.                                      */
/*-------------------------------------------------------------------------*/
else
  {lmbind(LMODEL, 0);
   cpack(0xffffff);
   CurrentObj = ListHead;
   while (CurrentObj != NULL)
     {for (i=0; i<CurrentObj->RotSteps; i++)
         {pushmatrix();
          rotate((long)(CurrentObj->RotIncr*i*10), AxisOfRotation);
          index = 0;
          for (j=0; j<CurrentObj->NumLines; j++)
             {bgnline();
              for (k=0; k<CurrentObj->NumPoints; k++)
                 {v3f(&(CurrentObj->Data[index]));
                  index += 3;
                 }
              endline();
             }
          popmatrix();
         }
       CurrentObj = CurrentObj->Next;
      }
  }
 return;
}
```

IN-34-CR

NASA National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
13		

4. Title and Subtitle

Computational Fluid Dynamics for Propulsion Technology

5. Report Date

3/17/92

6. Performing Organization Code

7. Author(s)

John P. Ziebarth

8. Performing Organization Report No.

10. Work Unit No.

9. Performing Organization Name and Address

University of Alabama in Huntsville
Huntsville, AL 35899

11. Contract or Grant No.

NAS8-36955/DO 85

13. Type of Report and Period Covered

FINAL 12/20/90-4/1/92

12. Sponsoring Agency Name and Address

Marshall Space Flight Center
National Aeronautics and Space Administration
Washington, D.C. 20546-0001

14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

This work involves the coordination of necessary resources, facilities and special personnel, to provide technical integration activities in the area of computational fluid dynamics applied to propulsion technology. Involved is the coordination of CFD activities between government, industry and universities. Current geometry modelling, grid generation and graghical methods will be established to use in the analysis of CFD design methodologies.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Computational Fluid Dynamics Numerical Grid Generation Propulsion	Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Un	Un	43	

NASA FORM 1626 OCT 86

ORIGINAL PAGE IS
OF POOR QUALITY