NRC Publications Archive
Archives des publications du CNRC

Design and implementation for 3D unsteady CFD data visualization using object-oriented MFC with open gl

Liu, P.

This publication could be one of several versions: author's original, accepted manuscript or the publisher's version. / La version de cette publication peut être l'une des suivantes : la version prépublication de l'auteur, la version acceptée du manuscrit ou la version de l'éditeur.

Publisher's version / Version de l'éditeur:

Computational Fluid Dynamics Journal, 11, 3, pp. 335-345, 2002

NRC Publications Record / Notice d'Archives des publications de CNRC:
https://nrc-publications.canada.ca/eng/view/object/?id=47efb2f7-b65b-453c-9e5e-77fa829ab8bd
https://publications-cnrc.canada.ca/fra/voir/objet/?id=47efb2f7-b65b-453c-9e5e-77fa829ab8bd

Access and use of this website and the material on it are subject to the Terms and Conditions set forth at
https://nrc-publications.canada.ca/eng/copyright
READ THESE TERMS AND CONDITIONS CAREFULLY BEFORE USING THIS WEBSITE.

L'accès à ce site Web et l'utilisation de son contenu sont assujettis aux conditions présentées dans le site
https://publications-cnrc.canada.ca/fra/droits
LISEZ CES CONDITIONS ATTENTIVEMENT AVANT D'UTILISER CE SITE WEB.

Questions? Contact the NRC Publications Archive team at
PublicationsArchive-ArchivesPublications@nrc-cnrc.gc.ca. If you wish to email the authors directly, please see the first page of the publication for their contact information.

Vous avez des questions? Nous pouvons vous aider. Pour communiquer directement avec un auteur, consultez la première page de la revue dans laquelle son article a été publié afin de trouver ses coordonnées. Si vous n'arrivez pas à les repérer, communiquez avec nous à PublicationsArchive-ArchivesPublications@nrc-cnrc.gc.ca.

National Research Council Canada | Conseil national de recherches Canada

Canada

Design and Implementation for 3D unsteady CFD Data Visualization Using Object-Oriented MFC with OpenGL

P. Liu [a,1],

[a]*National Research Council, Institute for Marine Dynamics, Box 12093, 1 Kerwin Place, St. John's, NF Canada, A1B 3T5*

Abstract

This article introduces an idea to improve the reusability, user friendliness and productivity of a general CFD code and describes the methods that were developed in design and implementation of a postprocessor. This component performed the required tasks to present colour-blended 3D unsteady data on moving bodies, as an application to unsteady CFD data visualization. This work provided general procedures to design and build a postprocessor for numerical modeling code, which is essential for the integration of a CFD code. This design and implementation work integrated the Microsoft Foundation Classes (MFC) in Visual C++ and OpenGL from SGI. Using the procedures and instructions presented in this article, various numerical modeling codes that were developed by research scientists, engineers and graduate students, could be modified with minor effort to become more valuable and productive because the processor adds data visibility, user-friendliness and hence the reusability of a CFD code.

Key words: CFD visualization, MFC, Visual C++, Object-oriented programming, GUI, and CAD graphics, panel methods, propeller

1 Introduction

With the development of web and multimedia technologies, physical quantities and computational geometry obtained from numerical modeling codes

[1] Email: Pengfei.Liu@NRC.CA

were represented more and more effectively and precisely. The pre- and post-processors became indispensable components for CFD (computational fluid dynamics) software packages and they could make these packages more user-friendly and more marketable. There was a huge number of codes developed by graduate students, scientists and engineers. Numerical kernels of these codes were usually highly sophisticated and they played important roles in numerical simulations in a particular field of interest. Their numerical capabilities were invaluable to scientific research and engineering design. However, without a user-friendly GUI and professional looking pre- and post-processor, reusability and marketability of the codes are poor. Many CFD codes that were written by graduate student were rarely used after the completion of their theses. Also, many codes written by researchers in one institution were not used when they moved to another. Therefore, the same institution or country no longer possesses the same numerical modeling capability as it did.

Many commercial CFD codes have nice-looking GUI and built-in pre- and postprocessor for mesh generation and data visualization. With the development of visualization tools and graphics technologies, it is no longer a difficult task for a numerical modeling scientist to build these features in a scientific computing code. A typical visualization tool that is invoke-able in the run-time is VISUAL2 for 2D visualization [1]. A 3D run-time invoke-able package VISUAL3 was also produced by Haimes [2]. The 3D version of the tool, pV3, was also available for a parallel computing environment under a parallel virtual machine (PVM). Another example tool is the visualization tool kit (VTK) which is a huge package [3]. Like the VISAUL2D and 3D, VTK also uses the OpenGL graphics libraries that were created by SGI, but VTK was written in C++ with source code. The VISAUL2D and 3D can be used by embedding them into a numerical modeling code and the VTK can be used as a separate visualization component. When Microsoft Foundation Class, the subset of the Visual C++ compiler, is used in the Windows environment, both of them can be embedded into the code. Using both the tools requires the effort to learn how to use them. Especially for the VTK, compilation of the source code in the development mode within Visual C++ is difficult. Most importantly, to

commercialize a CFD code, the owner of the CFD code has to consider the commercial limitations of using these tool kits and to monitor the changes of these limitations. In addition, VTK is overkill for a specialized CFD code that does not require extensive visualization analysis. As access to the OpenGL libraries is available in many C/C++ and Fortran compilers under both UNIX and Windows platforms, it enables the CFD programmers to write the visualization component of their own CFD code with a minor effort in learning the functionality of the OpenGL graphics libraries. Another advantage of developing CFD programmers' own visualization code is the flexibility of making modifications in terms of increasing functionality, enhancing capability and paralyzing the code for high performance computing.

The function of pre- and post-processors for a CFD application package is to generate and view the mesh, and to process and represent the computational output data. Mesh generation programmers know best the requirements to present the mesh, as do the CFD solver programmers to present their data. However, developing professional looking pre- and post-processors has been a challenge to many numerical modeling researchers, who were not professional computer graphics programmers, until the release of OpenGL by SGI. Various visual language compilers for the Windows platforms have been essential to creating the user-friendly and professional looking GUI for Windows based software applications. Equipped with these software development technologies and tools, design and implementation for pre- and post-processors became feasible. The following section discusses the design and implementation for such a software system.

2 Design of Application Functionality and Programming Environment

As mentioned in the previous section, a complete set of pre- and post-processors for CFD software includes grid generation, grid viewer, output data processing, representation of the processed data and capture of the motion pictures

for further playback and analysis. Grid generation is a different topic so only the mesh viewer and unsteady data presentation will be discussed.

The functionality of this visualization program covers three aspects. They are general requirements, requirements for the mesh viewer or the steady state data presentation, and requirements for unsteady CFD data presentation. These requirements are as follows:

2.1 General Requirements

- Highly user-friendly and very simple to use;
- Professional looking GUI that is integrated seamlessly with the operating system;
- Using as much memory as needed and no memory leaks (to avoid crashes);
- Resizable window with fast redraw;
- Combining both mesh viewer and unsteady CFD data visualization into one application so that the integrated code can be easily embedded into other applications;
- One simple data structure for the input file for both the geometry mesh viewer and unsteady CFD data visualization;
- Fast data access speed and efficient storage space usage;
- Applicable for both quadrilateral and triangular surface mesh grids;
- Conversion of both steady and unsteady mesh geometry to DXF file format to provide the possibility of interfacing with other graphics applications such as AutoCAD;
- Expandability of the code to add more functionality in the future;
- Generality. The code is applicable to other non-CFD applications.

2.2 Mesh View Requirements

- Wireframe mode;
- Surface mesh with hidden line removal;

- Solid modeling with lighting enhancement;
- Colour-blended polygon mode
- Option for designated geometry colours;
- Dynamic zooming, panning/translation and rotation about any one of the 3 axes by just a mouse button-click and dragging another, correspondingly;
- Smooth and fast graphics manipulation.

2.3 Requirements for Unsteady Data and Geometry Presentation

- Selection between discrete panel colours and blended colours for data presentation;
- Selection between different output data sets in the input file;
- Optional time interval adjustment between each time step to simulate reality or to maximize hardware speed capability;
- Inverse colour option;
- Sliding bar for colour density and range adjustment. This adjustment was necessary to obtain a contrast if there were one or more data points having a very high or low value. Without this adjustment, the body surface may have an almost uniform colour except for a few data points, resulting in a meaningless presentation;
- Optional view of motion geometry using either corner point blended colour, wireframe, solid modeling or hidden line removal without using the unsteady CFD data;
- Capture of high resolution and high colour gradient (up to million of colours or more) motion pictures for playback, frame-split and analysis by other multimedia utilities.

The design process was finalized in a cyclic and optimum process. The basic application functionality requirements were first determined and then the implementation analysis started in terms of feasibility and programming efficiency. The revised application functionality design was again compared with the user needs, and so on.

2.4 Programming Environment and Rationale

Based on the above functionality analysis, the programming environment and rationale are listed below.

- Application platform. In this case Windows was chosen. Accessibility and popularity were the advantages;
- Compiler option. There were a number of visual compilers available such as Visual C++, Borland C++ Builder, Visual BASIC, Visual J++, Visual Fortran, etc. A Microsoft compiler was chosen because of the availability of the compiler at work. Object-oriented Visual C++ was chosen because it had both the built-in OpenGL libraries and hardware controllability. Control of hardware was not used in the current version of the code but with C/C++, access of input and output devices can be easily implemented for further extension to add data acquisition and monitoring capabilities.
- Windows API or MFC? Visual C++ compiler offered two visual development tools to create executables. They were Windows Application Programming Interface and Microsoft Foundation Class. Using Windows 32-bit API has an advantage of many reusable code samples to perform various functionalities. They were ready to use with minor or no revision. MFC, on the other hand, has much higher efficiency in visual design and implementation. Classes were already in place to be implemented into the code by simply clicking the mouse buttons. It eliminated the need to create new generic classes for most applications. As there were not many sample codes that could be adapted directly, major revision to the Windows API code samples or creation of new code segments was required. Thus MFC was chosen as the programming environment.
- OpenGL libraries and conventional graphics programming. The odds and the ends in conventional graphics programming were the fundamentals for computational geometry and graphics. There was much literature in this category, including the works by Johnson [4] and Olfe [5]. However, OpenGL has a great advantage for graphics presentation, manipulation and especially

the productivity of programming. Therefore, OpenGL was chosen.

3 Programming and Implementation

This section describes the programming methods and techniques in the implementation. Three subsections in the following correspond to the requirements in 2.1, 2.2 and 2.3.

3.1 Implementation for the General Requirements

To create an executable application in Visual C++, three programming environments were available. They were 32-bit console programming (codes executed on a dark screen in terminal mode like an old DOS), 32-bit Windows API programming, and the Microsoft Foundation Class programming. In 2.4, MFC was chosen. As mentioned, it gave a better integration between the code and the operating system. In MFC, three types of GUI were available. They were single document interface (SDI), multiple-document interface (MDI) and dialog based interface. SDI was chosen because it met the requirements.

In Visual C++, the CSingleDocTemplate class generated a SDI application, which was a template class to define the type of the application object. In a SDI application, four major classes were created by the MFC [6] and they were:

- Application class, which was responsible for starting, initializing, running and closing the Windows application. This class was derived from the MFC CWinApp class with a virtual function defined from the base class, CWinApp.
- Window class, a derived class from CFrameWnd in MFC. This class did the creation and management of a window in the application. All window elements such as toolbars and menu buttons were the frame window components and were managed by the window class.
- Document class, which was a derived class of the CDocument class in MFC.

Data members and member functions were defined in this class to perform data retrieval and storage to and from disk, data processing, etc. GUI enabled functionalities such as file open, file save, etc. were inherited in this class without additional programming work. The frequently used MessageBox function for user interface and error handling did not work in this class. This was done by using the afxMessageBox function.

- View class, which was responsible for the client area inside the window. It was derived from the CView class in MFC. It was able to access data from the document class by using a pointer from the document to the data variables declared in the document class.

Interface between the window, the document and the view class was managed by the CSingleDocTemplate class. Class objects for the present work are shown in figure 1.

In figure 1, items in the document object were the functions to be called by the view class and by the functions or the objects of the frame window class. All the functions linking the buttons and toolbars on the frame windows were prototyped in the document class. The event-driven functions or dialog box objects were called or instantiated when a particular button was clicked at runtime. These event-driven class objects and functions may be defined inside either the view class or the document class. In this application, they were defined in the document class because of easy programming. This arrangement gave the flexibility of the programming capability and class interface ability.

An example operation to view the unsteady results is shown in the following steps:

- A user clicked the file open button and this invoked the GUI open file dialog by MFC;
- A file was selected by the user and the open file function was called to open the input file and process it. Dynamic array allocation was made in the document class and the pointer variable declarations were defined in the header file of the document class. The delete commands were put in the

destructor of the document class to avoid memory leaks.

- The user clicked the geometry button (this was necessary only if the position of the body needed to be initialized for viewing purpose). A dialog box object was instantiated and view parameters were set by the user. The values of the member variables in the dialog box object were passed dynamically to the document class member variables by using the function UpdateData(TRUE/FALSE).
- When the rotation button was clicked, the right mouse move function in the view class was called and the dragging distance of the right mouse button was captured to assign the value of the rotational angle. The rotation button was designed for the geometry view option but it was also for the initial geometry position of the moving body(s) for unsteady data visualization. A mouse movement invoked the corresponding mouse move functions in the view class. The distance was then captured as the amount of rotational angle that was used in the RenderScene function in the document class. This RenderScene function was called by the OnPaint function in the view class via a pointer.
- After a new dialog box object was instantiated and the class member variables could then be defined by the user or by default, then the view motion switch was enabled. The values of the member variables that were defined in the document class were passed to the view class to process unsteady data in the OnPaint function at each time step. The processed unsteady data were then sent to the RenderScene function that was called inside the OnPaint function of the view class, because the OnPaint function was called each time the screen was repainted. The processed unsteady data such as the instantaneous positions of the geometry and colour values on each panel were stored in the DRAM for fast access.
- A checkbox in the GUI was used to enable the capture of the moving pictures. The GUI dialog box for movie capture was created by the vfw32.lib, which was provided by Microsoft in its Windows operating systems.
- View geometry and view motion may be enabled alternatively. Options for discrete and blended colours, and viewing modes such as wireframe, hid-

den line removal and solid modeling were made available by adjusting the parameters controlling the OpenGL functions. The initial position of the unsteady geometry was based on the first frame of the movie, which was created for the first time step. This initial position was to be manipulated by the user for desired view point as mentioned in the third step above.

Although MS Visual C++ version 6.0 included OpenGL libraries, it needed a few modifications to the MFC code. There were four steps to set up the MFC to work with OpenGL and they were:

- Including OpenGL header files and libraries;
- Setting Windows styles for rendering context creation;
- Creating the rendering context and making it current; and
- Creating a function OnDestroy to release the rendering device context when Windows was to be closed.

Step by step instructions to the modification were given by Oursland[8].

A binary file structure was designed for the input file for fast access and storage efficiency. The structure of the input file outputted from a CFD numerical model is shown in figure 2.

In figure 2, i was a 4-byte integer standing for the ith time step. For surface geometry view or steady data visualization there was only one time step, $I = 1$. For unsteady data visualization, $i = n...I$, where I was the number of the last time step. The starting time step was an arbitrary integer n so that the number of total time steps was then $I - n + 1$. Integer J was 4-byte-long standing for the number of panels on the object's geometry at the ith time step. Subscript j indicates the jth panel, $j = 1...J$. The value of J, i.e. the number of total body panels, was designed to be a variable for different time steps. This allowed the possibility of losing or adding panels during the numerical computation for explosion or adaptive grid arrangement.

The values of C_{ij} and D_{ij} were the first and the second physical quantities. These 4-byte floating-point numbers could be, for example, a continuous pres-

sure value and an ON/OFF cavitation index on the panel at the ith time step at the jth panel. The 12 values of x_{1ij} to z_{4ij} were to define the 4 corner points of the jth panel at the ith time step. Four points represented a quadrilateral panel. For triangular panels, the last point was set the same as the third point so the data structure of the input file did not need a change. This data structure was theoretically applicable for any number of moving objects and any number of data sets, provided that the physical memory of a computer was sufficient.

To achieve interface ability with other CAD packages, functions to convert both the geometry and the unsteady moving surfaces to DXF files, were prototyped in the document class. One DXF file was created for each time step. For unsteady data, I files were created and the time step index i was used to flag the file names.

3.2 Implementation of the Functionalities for the Mesh Viewer

The requirements for the mesh viewer were met by incorporating the OpenGL libraries and the functions in the view and document class of MFC. Each panel was treated as a polygon in OpenGL[9]. For wireframe, the following code segment in the RenderScene function of the document class was used:

```
glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
```

and for blended colour panel representation, the code segment was like this:

```
glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
```

For blended colour, the colour for each corner point of a panel needed to be user-inputted.

For solid surface rendering, the following OpenGL function was employed:

```
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, PanelSurfaceColor);
```

The colour array defined in the code, PanelSurfaceColor, took different values based on the user's configuration for unsteady data visualization. Mesh viewer and unsteady visualization viewer jointly worked to achieve various visualization options.

For hidden line removal, *Polygon Offset* method was used [9]. In the polygon offset method, each panel was drawn twice, i.e. each panel was first drawn with the wireframe edges. Colour filled panels then overlapped these wireframe polygons. The filled panels were slightly nudged further from the viewer using the glPolygonOffset function. In the code, only one time step needed to be coded. The time step loop was put in the OnPaint function of the view class. The arrays to store the geometry and CFD data were in the document class. At each time step, OnPaint function called the RenderScene function to perform the graphics drawing work on the client area.

Rotation angles about the three axes were taken from the dragging distance of the right mouse button. The dynamically exchanged values of the angles were used by the OpenGL rotation function after a corresponding mouse move function in the view class was called. For each minimum moving distance, the OnPaint function in the view class was called and it then called the RenderScene function.

The code samples for rotation about an axis and for panning (translation) can be easily found in a number of OpenGL examples from the SGI's web site [7]. However, zooming (magnification) was not seen to be discussed yet. Zooming in and out requires a smooth transition and change. A method for zooming was created and it is shown in figure 3, which gave a magnification factor from about 10^{-3} to 10^{3} times.

The double buffering frame method was employed for fast graphics display. The OpenGL function for this was:

```
SwapBuffers(dc.m_ps.hdc);
```

This function was called after the RenderScene function being called inside the OnPaint function as suggested by Oursland [8].

4 Implementation for the Unsteady Data and Geometry Visualization

For discrete panel colour viewing, the physical quantities on each panel were mapped onto the hexagonal hue region with the maximum and minimum data values to correspond to the hue angles of 0^0 to 60^0. This hue value of each physical quantity was then converted to RGB colour scheme. A description of CAD graphics colour calculation and application along with a C function to convert hue to RGB was given by Olfe [5]. A discrete colouring scheme was desired when an individual panel has the physical quantities being binary numbers such as ON/OFF. An example of this would be the cavitation property of a panel on a propeller or hydrofoil surface.

For physical quantities that were continuous across body surfaces, discrete colour gave an undesired view. Blended colouring scheme was a solution to this problem. For blended colour presentation, the colour gradient at each panel was achieved by blending the 4 colours at each point of a quadrilateral or a triangular panel. The averaged physical quantities at each node point, i.e. at each corner point of a panel were first calculated by interpolating the physical values of the adjacent panels of each node. For triangular panel arrangement, there were usually 3 adjacent panels around a nodal point and for a quadrilateral panel, there were usually 4 adjacent panels to interpolate. However, at nodal intersection of two bodies such as between the hub and the blade's trailing and leading edges, this node could have as many as 6 adjacent panels. Six adjacent panels were coded to participate in the interpolation.

The interpolation, quantity mapping onto the hue values, colour conversion, and finding the panel normal vector (for solid surface modeling view) took a fair amount of computing resources. These operations were performed only when both a new file was opened and the view motion button was clicked, or

one of the values was changed by the user via sliding bars, to save the CPU time.

Currently, the code has two sets of output physical quantities to choose from for each view motion object. The C_{ij} and D_{ij} were used to store the instantaneous pressure coefficient and the cavitation property (yes or no) at each panel. For marine propellers in unsteady, non-uniform inflow, the pressure and cavitation visualization is important to observe force fluctuation, material damage location and cyclic behavior of cavitation.

For pure motion animation, the physical quantities were ignored in the code. Only the geometry at each time step was sent to the RenderScene function. In this case, the appearance of an object was controlled by the member variables in view motion dialog box class, which was instantiated in the document class.

Colour density and range adjustment were found necessary for colour contrast and gradient. A few very big or very small values in the CFD data array caused undesired plot. This colour gradient adjustment was made by the user to customize the minimum and maximum cut-off values in the input data. Two sliding bars were implemented for these cut-off values.

A timer was set in the frame window class and the event driven associated function was prototyped in the document class for easy data access. The pace of displaying the unsteady motion images was controlled by the timer for fine visualization and hardware performance. Each non-duplicated frame of the displayed motion pictures was saved into DRAM from the screen buffer and then transferred to the hard disk.

5 Results and Discussion

As in most cases, using binary file saves about 50% of hard disk storage and about four times file access speed. Quick file retrieval was the major gain from this approach because disk storage was no longer a big problem. For a 4-blade

surface piercing propeller of about three thousand panels along with 180 time steps, each binary input file size was about 30 Mb. The size of each captured AVI file for one viewpoint ranged from 100 to 200 Mb, without compression. For a prediction of thrust and torque coefficients of a propeller at each advance ratio, each data point along these two continuous curves produced one such binary file.

Visual design of this visualization application is shown in figure 4. The procedures for the visual design of a MFC program may be found in some "21 days" C++ books for beginners. Caution was taken when adding new classes: the added new classes were not easily deleted by simply deleting the class file and the header file. Links to other classes and visual components had to be deleted accordingly.

Creation of the progress bar for file retrieval and for operations to prepare motion visualization was written. An easy alternative was to use the progress bar included in the Visual C++ add-on components. A tooltip feature was implemented by adding a tooltip component that came with the Visual C++ compiler 6.0. This component enabled a simple tooltip to be displayed at the mouse cursor position when the mouse hovered over the menu button or toolbar and gave a more detailed explanation at the lower left corner of the application window.

Figure 5 shows the implemented visual design of the view geometry dialog box object. To view geometry with a blended colour, the blended colour switch was turned on. For blended colour view geometry option, each corner point of the panels had 20 colours to choose from. These 20 colours in the dialog box were also available for wireframe, solid surface rendering and hidden line removal modes. Figure 6 shows visual design of the colour selection subdialog box.

A propeller-rudder-nozzle assembly is shown in figure 7. The surface meshes of figures 7 to 10 were generated by a panel method code pre-processor[10]. The geometry assembly in figure 7 was used for a hydrodynamic characteristics analysis of multi-body, multi-path, unsteady lifting flow around a marine

propeller[11].

Figure 8 shows an ice-class propeller with an ice blockage under interactive induced flow or ice-propeller contacting flow. CFD results for this kind of geometry shape were obtained for studying both the structural and hydrodynamic characteristics of ice propeller interaction[12].

Figure 9 shows the solid modeling of a DTMB (David Taylor Model Basin) propeller P4119 with a customized rudder. The lighting configurations were fixed in the code.

A highly skewed, 3-blade DTMB propeller P4679 is shown in figure 10. Wireframe view gives more detailed information on blade sectional shape.

All colour selections, geometry formats, rotations, translations and magnifications were designed and implemented for simplicity of use. These operations can be achieved by just clicking and dragging the mouse buttons. The GUI implementation to set the view motion parameters for the view motion dialog box class is shown in figure 11. The view geometry dialog box class was responsible for the initial viewpoint of the moving objects.

Figure 12 shows one of the instantaneous positions of a simplified 4-blade propeller with colour blended pressure coefficient distribution on the suction side of the blade. Captured motion picture frames were displayed by the Window Media Player that was built-in with the Windows operating systems.

As the code was a Window API application, pictures on the client area or the full screen were able to be copied into a clipboard and pasted to other word processor or photo software applications for image editing or publication.

Hardware requirements for the application varied with the nature of the input file. Normally a Pentium Pro or better microprocessor with a 12-bit or larger video card was enough. Sufficient dynamic random access memory (DRAM) was required. The required size of the physical memory was

$$DRAM_{size} = NTPBody \cdot (12 + 2) \cdot NTSM \cdot 4 \cdot 2 = 112 \cdot NTPBody \cdot NTSM,$$

where $NTPBody$ was the number of total panels of the body or bodies, and $NTSM$ was the number of total time steps. Numbers 12, 2, 4 and 2 were based on 3 coordinates of each corner point of a four-corner-point panel, 2 physical quantities for each panel, 4-byte long floating point number of each of the above values and rooms to hold other operational arrays with a factor of two. A body of 10,000 panels with 180 time steps required about 200 Mb of DRAM.

6 Conclusion

Procedures and methodologies to design and implement unsteady computational 3D visualization were described. A CFD visualization code development was given as an example. The procedures and methods were general enough to be applicable for other kinds of computational geometry and data visualization.

Equipped with the powerful and easy-to-use object-oriented visual programming tool MFC and OpenGL, it is feasible for numerical modeling researchers and engineers to write a pre- and post-processor for their numerical codes. This capability gives numerical modeling codes added value in terms of reusability and marketability.

7 Acknowledgments

The author thanks the National Research Council of Canada for its support. Mr. Derek Yetman's proofreading is also appreciated.

References

[1] Haimes, B. and Giles, M. (1992). *VISUAL2 User's & Programmer's Manual*, http://raphael.mit.edu/visual2/user2.ps, 23p.

[2] Haimes, B. (1998). *VISUAL3 User's & Programmer's Manual*, http://raphael.mit.edu/visual3/user3.ps, 60p.

[3] Schroeder, W., Martin, K. and , Lorensen, L. (1998). *The Visualization Tool Kit, An object-oriented approach to 3D graphics*, 2nd Edition, Prentice Hall PTR, 645p.

[4] Johnson, N. (1987). *Advanced Graphics in C*, McGraw-Hill, New York, 670p.

[5] Olfe, D.B. (1995). *Computer Graphics for Design: From Algorithms to AutoCAD*, Prentice Hall, Englewood Cliffs, New Jersey, 544p.

[6] Horton, I. (1997). *Beginning Visual C++ 5*, WROX Press Ltd., Birmingham, 1052p.

[7] SGI, "The Industry's Foundation for High Performance Graphics," http://www.opengl.org/

[8] Oursland, A. "Using OpenGL in Visual C++ Version 4.x," http://devcentral.iftech.com/learning/tutorials/mfc-win32/opengl/

[9] Woo, M., Neider, J., Davis, T., Shreiner, D. (1999). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*, Third Edition, Addison-Wesley, Reading, Massachusetts, 730p.

[10] Liu, P., Bose N. and Colbourne, B. (2001). "Automated Marine Propeller Geometry Generation of Arbitrary Configuration and a Wake Model for far Field Momentum Prediction", International Ship Building Progress, Vol. 48, No. 4., pp. 358-383.

[11] Liu, P. and Bose, N. (2000). "Hydrodynamic Characteristics of a Screw-Nozzle-Rudder Assembly," Journal of Computational Fluid Dynamics of Japan, vol. 8.

[12] Liu, P., Doucet, J.M., Veitch, B., Robbins, I., and Bose, N. (2000). "Numerical Prediction of Ice Induced Hydrodynamic Loads on Propellers Due to Blockage," Oceanic Engineering International, vol. 4, no 1.

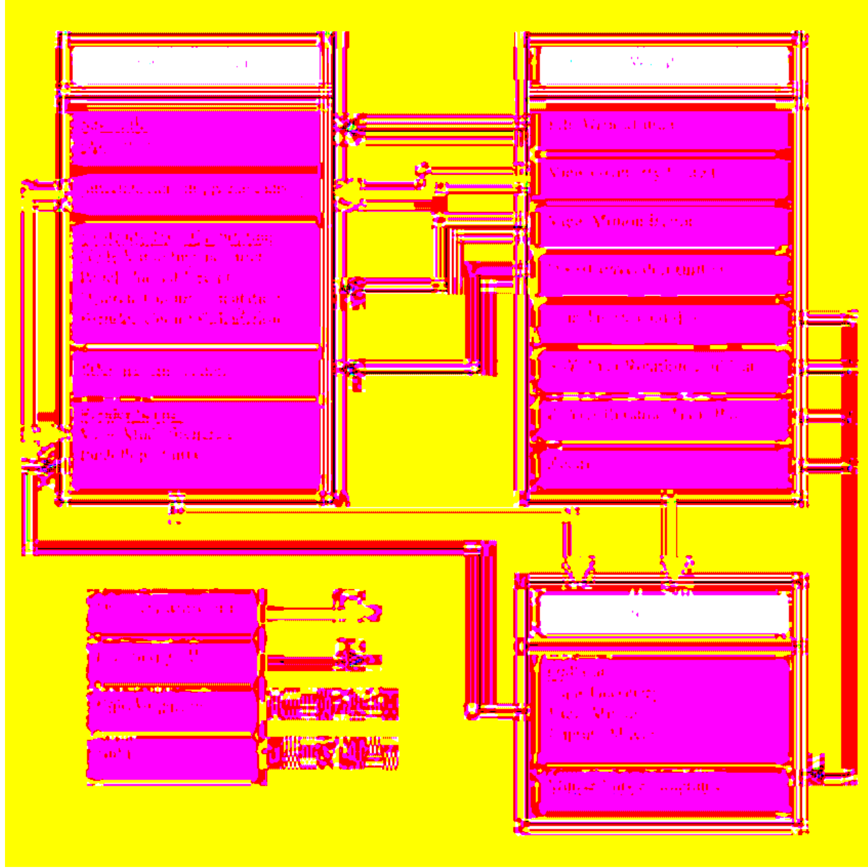

Fig. 1. Class object operation diagram.

i	J_i												
X_{1i1}	Y_{1i1}	Z_{1i1}	X_{2i1}	Y_{2i1}	Z_{2i1}	X_{3i1}	Y_{3i1}	Z_{3i1}	X_{4i1}	Y_{4i1}	Z_{4i1}	C_{i1}	D_{i1}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
X_{1ij_i}	Y_{1ij_i}	Z_{1ij_i}	X_{2ij_i}	Y_{2ij_i}	Z_{2ij_i}	X_{3ij_i}	Y_{3ij_i}	Z_{3ij_i}	X_{4ij_i}	Y_{4ij_i}	Z_{4ij_i}	C_{ij_i}	D_{ij_i}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
X_{1iJ_i}	Y_{1iJ_i}	Z_{1iJ_i}	X_{2iJ_i}	Y_{2iJ_i}	Z_{2iJ_i}	X_{3iJ_i}	Y_{3iJ_i}	Z_{3iJ_i}	X_{4iJ_i}	Y_{4iJ_i}	Z_{4iJ_i}	C_{iJ_i}	D_{iJ_i}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
I	J_I												
X_{1I1}	Y_{1I1}	Z_{1I1}	X_{2I1}	Y_{2I1}	Z_{2I1}	X_{3I1}	Y_{3I1}	Z_{3I1}	X_{4I1}	Y_{4I1}	Z_{4I1}	C_{I1}	D_{I1}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
X_{1Ij_I}	Y_{1Ij_I}	Z_{1Ij_I}	X_{2Ij_I}	Y_{2Ij_I}	Z_{2Ij_I}	X_{3Ij_I}	Y_{3Ij_I}	Z_{3Ij_I}	X_{4Ij_I}	Y_{4Ij_I}	Z_{4Ij_I}	C_{Ij_I}	D_{Ij_I}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
X_{1IJ_I}	Y_{1IJ_I}	Z_{1IJ_I}	X_{2IJ_I}	Y_{2IJ_I}	Z_{2IJ_I}	X_{3IJ_I}	Y_{3IJ_I}	Z_{3IJ_I}	X_{4IJ_I}	Y_{4IJ_I}	Z_{4IJ_I}	C_{IJ_I}	D_{IJ_I}

Fig. 2. The input binary file structure.

```
if (m_RightButtonDown && m_MagnificationStatus)
{
 CPPPPDoc* pDoc = GetDocument();
 CPoint scale = m_RightDownPos - point;
 m_RightDownPos = point;
 m_ScaleTemp += scale.y; //zoom when mouse moves up/down
 if(m_ScaleTemp<=-999)
 m_ScaleTemp=-999;
 if(m_ScaleTemp>=999)
 m_ScaleTemp=999;
 if(m_ScaleTemp>0)
 pDoc->m_ScaleFactor=(1.0-(GLdouble)m_ScaleTemp/1000.0);
 else
 pDoc->m_ScaleFactor=1.0/(1.0+(GLdouble)m_ScaleTemp/1000.0);
}
```

Fig. 3. Implementaiton of the zooming algrithm.

Fig. 4. Visual design of the software.

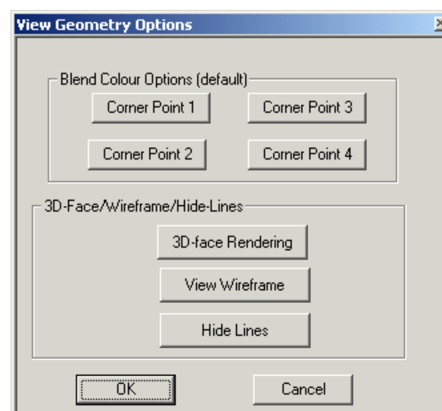

Fig. 5. Visual design of the view geometry dialog box.

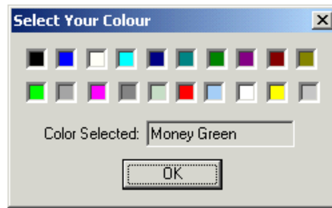

Fig. 6. Visual design of colour selection dialog box.

Fig. 7. Blended colour geometry view of a complex geometry combination of marine propeller, nozzle and rudder assembly.

Fig. 8. Geometry view of an ice-class propeller and with a contour-fitted ice wall. The display mode was hidden line removal.

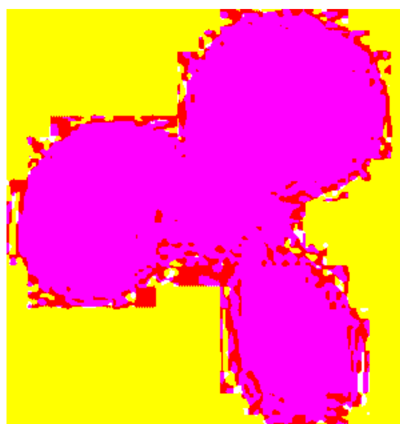

Fig. 9. A surface solid modeling of a DTMB 3-blade propeller P4119 with lighting configurations in OpenGL.

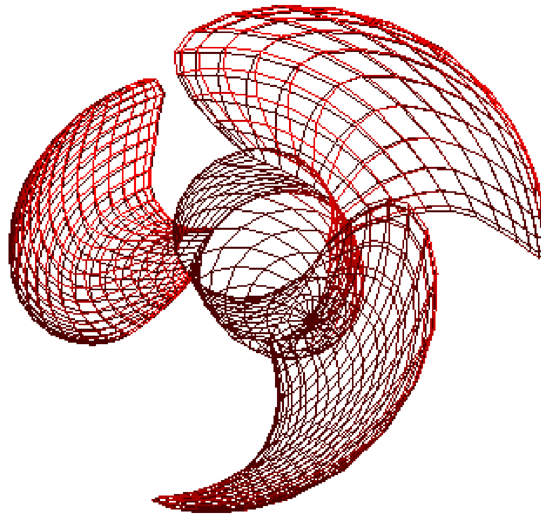

Fig. 10. A wireframe view of a DTMB 3-blade, highly skewed propeller P4679.

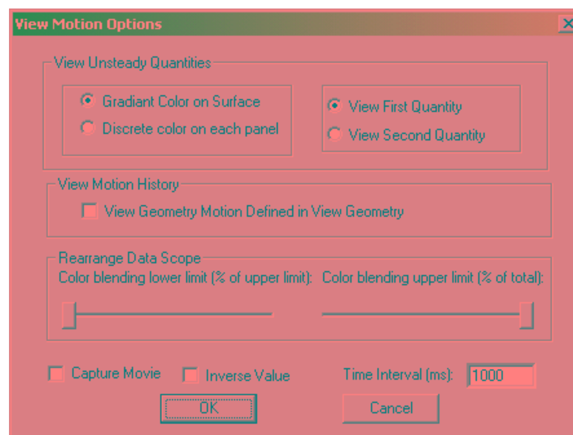

Fig. 11. The implemented visual design of the view motion dialog box.

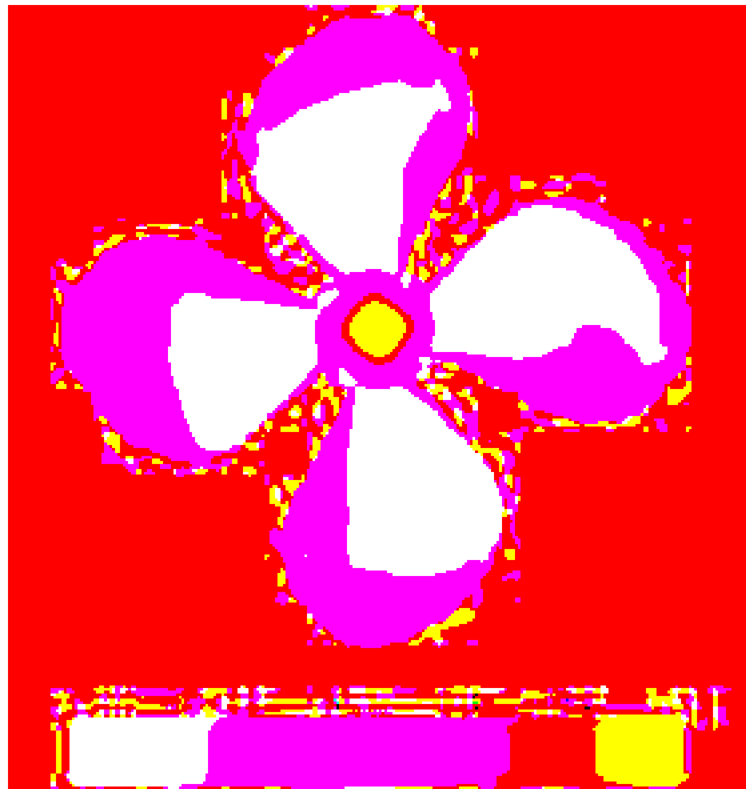

Fig. 12. Colour blended representation of pressure distribution of a 4-blade high speed propeller at one of the time steps in unsteady flow.