Modeling, Simulation and VR Visualization of Brain Aneurysm Blood Flow

UCLA Computer Science Department
Technical Report 960018

Michael Ray Harreld

July 1996

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Brain aneurysms are now clinically treated using endovascular procedures[GVSM91, GVDD91]. To plan these procedures, physicians use angiography to view projectional images of anatomy and bulk blood flow. There are no tools available, however, for visualizing detailed blood flow velocity and stress or predicting the effects of specific treatments. Availability of such tools would reduce the danger of aneurysm surgery and advance knowledge of aneurysm hemodynamics and pathology

Computer based technologies such as image processing, flow simulation and visualization, and virtual reality provide a potential for beginning to solve this clinical problem, thus increasing the quality of health care. It should be possible to use image processing to extract basic aneurysm information such as geometry and bulk flow from medical images. Using a physiological model, this data could then be used to numerically simulate detailed blood flow in an aneurysm. Finally the three-dimensional pulsatile blood flow data could be interactively and immersively visualized using a combination of flow visualization and virtual reality techniques. Starting from the vision of a clinically useful tool for immersively exploring aneurysm blood flow and simulating treatment, this dissertation describes the issues encountered and progress achieved in the design, implementation, and evaluation of the "Virtual Aneurysm."

1.1 Intracranial Aneurysms

Aneurysms are saccular dilatations of the vascular wall[Osb80](see Figure 1.1). Intracranial aneurysms are usually located at the base of the brain in the circle of Willis. The circle of Willis is an arterial ring that connects the two internal carotid systems (see Figure 1.2.) It has been reported[Osb80] that approximately 30% of all intracranial aneurysms occur on the anterior communicating artery, 20–30% between the interior carotid and posterior communicating arteries, 20% from the middle cerebral artery bifurcation, and 15% from the basilar tip. If aneurysms rupture (Figure 1.3) they can cause a catastrophic hemorrhage, often resulting in death. It has been found that 20–25% of all intracranial hemorrhages are caused by intracranial aneurysms[PLZ79]. Furthermore, intracranial aneurysms are responsible for the majority of all non traumatic subarachnoid hemorrhages[PLZ79, Fox83].

The true incidence of intracranial aneurysms is uncertain. From general autopsies, the reported occurrence of aneurysms varies from 0.2 to 7.9%. The reported percentage has been found to vary with investigator experience and with the purpose of autopsy. Roughly half of all aneurysms found during autopsy have ruptured[Fox83]

Intracranial aneurysms are classified into several categories[PLZ79]. The most commonly occurring type is the saccular aneurysm which comprises 66–90% of all aneurysms (see Figure 1.1.) These dilatations are found near an arterial bifurcation in roughly 75% of all cases. Otherwise they are found as lateral wall aneurysms on straight or curved vessels. Saccular aneurysms are often called berry aneurysms because they resemble berries growing on a branch. Unruptured aneurysms are typically found to have diameters from 2 to 10 mm. These smaller aneurysms often have thick necks and thin transparent walls. Aneurysms that have ruptured are usually larger, ranging from 6 to 50 mm in diameter. These aneurysms are sometimes calcified and partially filled with blood clots[PLZ79].

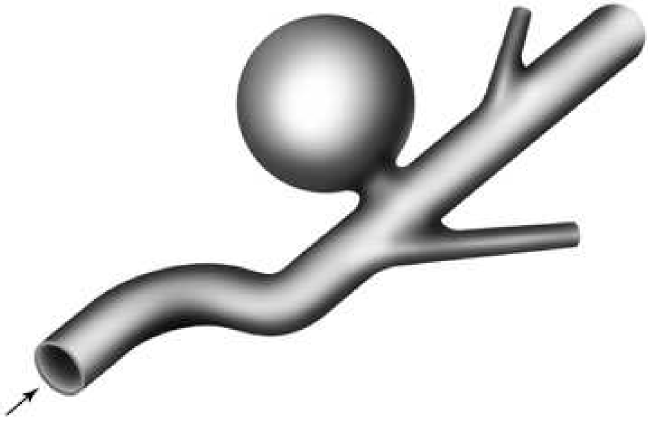

Figure 1.1: Saccular aneurysm.

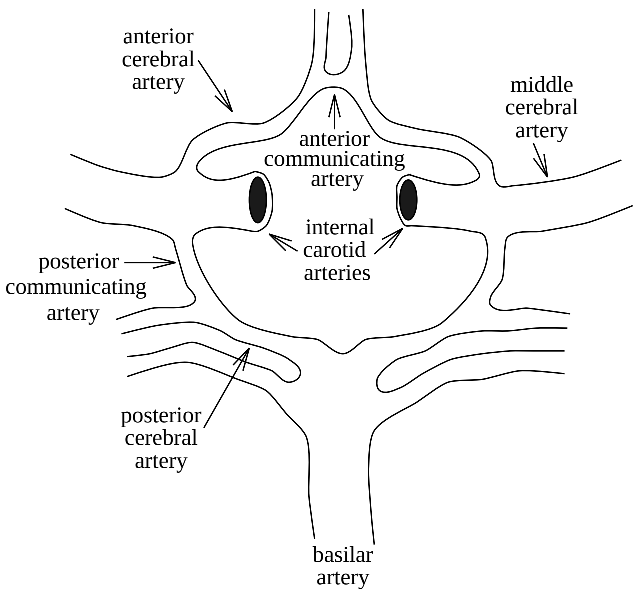

Figure 1.2: Schematic of circle of Willis showing major arteries.

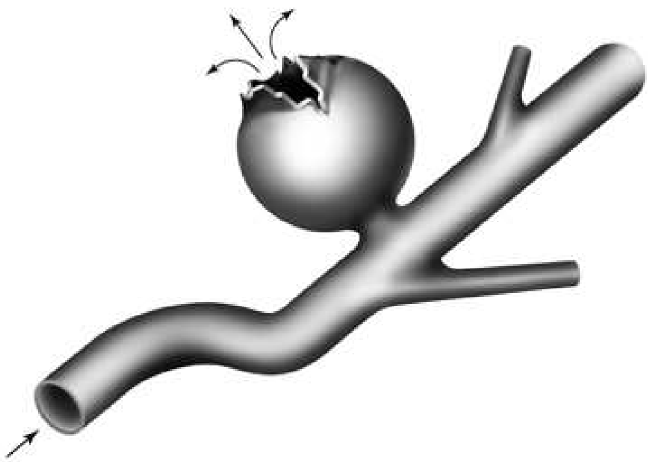

Figure 1.3: Lateral wall aneurysm rupture.

There are other less common varieties of arterial dilatations classified as aneurysms. The fusiform or atherosclerotic aneurysm appears as a region of artery with a local increase in diameter (see Figure 1.4.) Mycotic aneurysms are formed by bacteria or fungi that lodge in the artery wall. Dissecting aneurysms occur when the arterial wall is separated forming a pocket that inflates with blood.

In saccular aneurysms, the neck (also called the ostium) refers to the connecting region between the aneurysm and parent artery. The neck is further divided into proximal (upstream) and distal (downstream) regions. The center area of the aneurysm is referred to as the fundus, while the dome refers to the top of the aneurysm, i.e. the area farthest from the neck. Figure 1.5 illustrates the regions and terms in a lateral wall aneurysm. The inflow region is the area in the neck where flow passes from the parent artery into the aneurysm, and the outflow region is where blood exits the aneurysm and rejoins the arterial flow. Flow patterns in arteries and aneurysms are referred to as either laminar, transitional, or turbulent. In laminar flow, flow path lines are smooth, regular, and devoid of mixing. With turbulent flow, path lines are chaotic, irregular, and tortuous. Turbulent fluids exhibits extreme flow mixing and many small eddies. Transitional flow occurs between laminar and turbulent conditions. Minor instabilities and vortex formation are typical evidence of transitional flow[RC85, And84].

1.1.1 Causes and Detection

There are two major theories on the cause of intracranial aneurysms. The congenital theory contends that aneurysms are caused by an inherited weakness in the vascular wall. The degeneration theory holds that aneurysms are caused by acquired factors. Hemodynamics and atherosclerosis effects are often cited as mechanisms of the degeneration theory.

Research is ongoing to determine the cause of aneurysms and to discover the mechanism of their growth. It is known that hemodynamic factors are linked to the etiology of aneurysms, rather than inherited factors[Fox83]. Pathogenesis, growth, thrombosis, and rupture of intracranial aneurysms have been directly related to the effects of hemodynamic forces[SGR92]. In addition to weakness in the vascular wall and hypertension, local hemodynamic features have been found to be partially responsible for aneurysm growth and rupture. Clinically this can be shown by the large number of aneurysms found in the arterial loop called the circle of Willis at the base of the brain[LSPR87].

It is generally difficult to diagnose an aneurysm before rupture occurs. Sometimes symptoms such as headaches, dizziness, blindness, and neurologic disorders will manifest themselves and prompt testing[PLZ79]. Angiography is the primary means of locating and identifying intracranial aneurysms, and can provide the

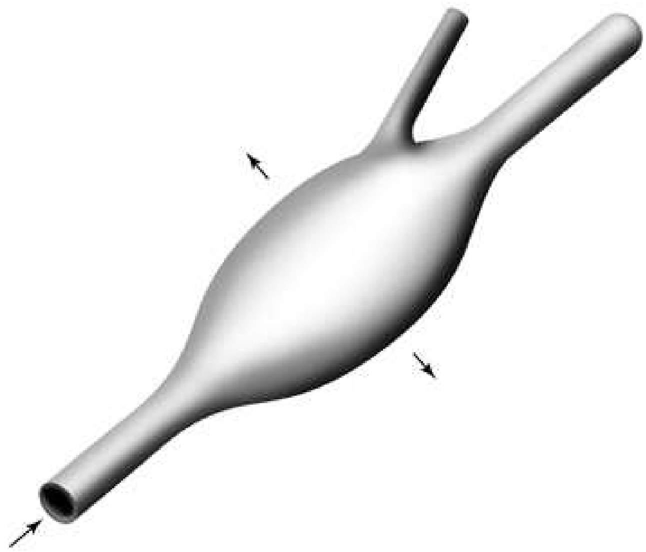

Figure 1.4: Fusiform aneurysm.

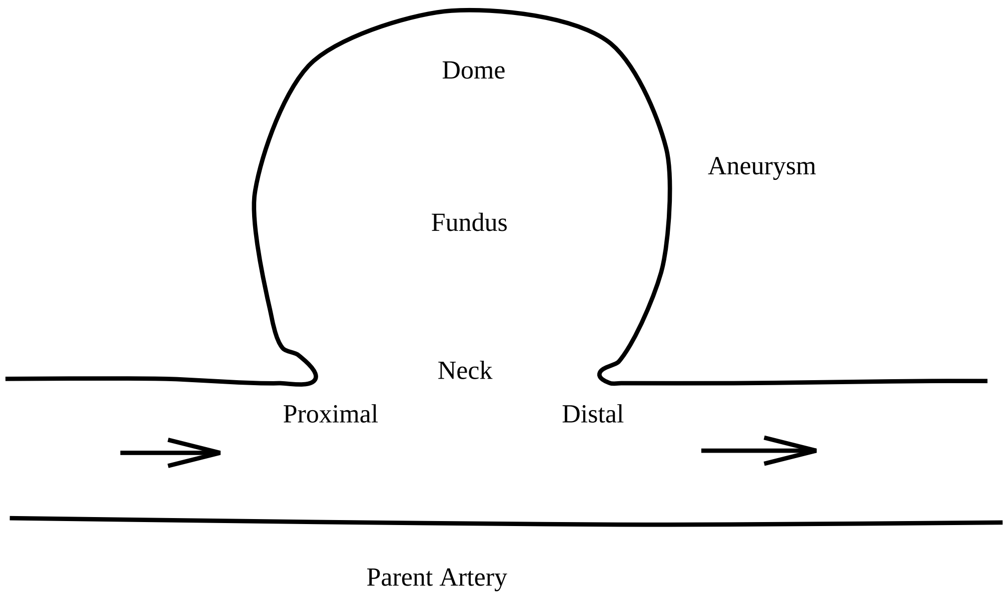

Figure 1.5: Schematic of lateral wall aneurysm with anatomic terms.

Researcher	Method	Pulsat.	Blood	Rigidity	Config.
Niimi[NKS84]	in vitro	puls.	suspension	rigid	lat. sac.
Liepsch[LSPR87]	in vitro	steady/ puls.	Newt./ non-Newt.	rig+elast.	lat. sac.
Perktold[PKH+88]	2D numer.	puls.	Newt.	rigid	bif. sac
Graves[GSPR92]	in vivo	puls.	canine	elastic	lat. sac.
Strother[SGR92]	in vivo	puls.	canine	elastic	lat. sac./ bif. sac.
Gonzalez[GCOM92]	2D numer.	puls.	non-Newt.	rigid	lat. sac.
Low[LPR93]	3D numer.	puls.	non-Newt.	rig.+elast.	lat. sac.
Kumar[KN95]	2D numer.	puls.	suspension	rigid	fusiform
Burleson[BST95]	2D numer.	steady	Newt.	rigid	lat. sac.

Table 1.1: Summary of aneurysm blood flow simulation efforts.

following information[PLZ79]:

1. Identification and location
2. Size and number
3. Neck configuration
4. Evidence of spasm
5. Relation to other arteries

Due to the diffusion of the contrast bolus and the two-dimensional nature of angiograms, it is difficult to determine hemodynamic parameters from angiography. Intravascular methods such as Doppler probes allow some measurements to be made of hemodynamic structure. Unfortunately, use of these invasive methods in a clinical setting adds additional risk to either interventional or diagnostic procedures.

1.1.2 Blood Flow

Research has been done to investigate blood flow in intracranial aneurysms. Studies have been made based on glass models, cardiometric measurements, and computer simulations of lateral wall aneurysms. Table 1.1 summarizes the methods and results of some of the major attempts at aneurysm blood flow simulation.

Liepsch et al.[LSPR87] used glass and silastic models with Newtonian and non-Newtonian perfusion fluids to simulate blood flow in aneurysms. It was found that inflow occurred at the distal lip and was directed towards center of the fundus. Outflow was found to occur at the walls of the fundus. They suggested that flow and stress are directly related to pulsatility and viscosity. Based on in vitro experiments of aneurysms on curved vessels, Niimi, Kawano, and Sugiyama[NKS84] found that curved arteries set up significant secondary flows (rotational flows in a plane normal to the general flow trend).

Perktold et al.[PKH+88] numerically simulated two-dimensional pulsatile flow in a saccular aneurysm at an arterial bifurcation. They found relatively slow rotating flow within the aneurysm with a dependence on arterial speed and aneurysm geometry.

Graves et al.[GSPR92] used angiography to study blood flow in dog aneurysms. They found that blood entered the aneurysm in a fast jet in the distal region, traveled in a slow vortex, and then exited more slowly in the proximal region. Blocking or deflecting the inflow or filling the aneurysm with embolic material was found to lead to thrombosis. Position and shape of the embolic material with respect to the inflow zone was determined to be critical to successful embolization.

Strother, Graves, and Rappe[SGR92] further explored blood flow in dog aneurysms with angiography and color Doppler methods. It was found that stagnation occurred in the vortex. The inflow zone was comprised of a central fast jet and a surrounding slow region. Disturbed laminar flow was found in the artery just below the neck opening. The flow in aneurysms was found to be not turbulent but predictable, in the laminar and transitional ranges. The hemodynamic stresses were found to vary between different aneurysms.

Gonzalez et al.[GCOM92] constructed a computer simulation of blood flow in a lateral wall aneurysm. The simulation was based on a two-dimensional model with pulsatile blood flow. Blood formed vortices in the aneurysm that reversed direction during the diastolic period. By modeling the aneurysm as two-dimensional, the assumption was made that secondary flow was negligible. This assumption contradicts the findings of Niimi et al[NKS84].

Low et al.[LPR93] numerically solved the Navier-Stokes equations describing incompressible, pulsatile, three-dimensional non-Newtonian flow in rigid and distensible lateral wall aneurysms on straight and curved arteries. They found that aneurysm inflow occurred at the downstream lip and outflow along the walls. A complex velocity field was observed within the aneurysm, with aneurysm speeds an order of magnitude below arterial speeds. Aneurysm wall shear stress was found to be low, except at the upstream and downstream wall regions, where strong shear stress concentrations were found. In the distensible model, aneurysm flows were higher than in the rigid model, but still low compared to arterial speeds.

Kumar and Naidu[KN95] modeled and numerically simulated pulsatile suspension flow through an axisymmetric fusiform aneurysm. The aim of the study was to study the effects of suspended particles (modeling blood cells) thus providing a more accurate simulation of blood flow dynamics. The particles were found to dampen velocity and wall shear stress.

Burleson, Strother, and Turitto[BST95] numerically simulated steady state, two-dimensional flow in rigid lateral wall aneurysms with half-spherical, spherical, and pear-shaped configurations. Maximum aneurysm shear stresses were found at the downstream neck. It was found that shear stress and flow were dependent on aneurysm size and shape.

1.1.3 Endovascular Treatment

The availability of highly controllable, soft microcatheters has enabled radiologists to navigate into and beyond the circle of Willis to provide intravascular treatment in areas within the brain that are difficult to access surgically[VDLD89]. Intracranial aneurysms can be occluded by inserting small metallic coils or balloons into the aneurysm.

Endovascular aneurysm treatment using metallic coils has been pioneered at the UCLA Medical Center[GVSM91, GVDD91]. A microcatheter with enclosed guide wire is carefully tracked using angiography and navigated through a groin artery of the conscious patient, up through the carotid, into the circle of Willis, and finally into the aneurysm (Figure 1.6.) The guide wire is then removed and replaced with a very soft detachable platinum coil at the end of a stainless steel delivery wire. The coil is inserted into the aneurysm to fill it and reduce blood flow (Figure 1.7.) A small direct electric current is passed through the coil to accelerate thrombosis and electrolytically detach the coil. After coil detachment, the delivery wire and microcatheter can be removed. Angiography is used throughout the procedure to monitor the relative locations of the aneurysm, microcatheter, and platinum coil. Depending on the size and configuration of the aneurysm, multiple coils of different sizes can be inserted.

1.2 Flow Visualization

1.2.1 Visualization Tools

Much work has been done to design tools for three-dimensional flow visualization. These tools help perform different visualization functions such as understanding flow at a point, on a surface, or in a field.

For in depth investigation of a particular flow area, point probes can be used. A simple velocity arrow is often used for this purpose. An expansion on this idea is a flow probe tool developed by de Leeuw and van Wijk[dLvW93]. The tool is a complex icon with separate elements to display velocity, curvature, acceleration, rotation, shear, and convergence.

Two-dimensional flow visualization can be used in three-dimensional flow to view cutting planes, body surfaces, or isosurfaces. A cutting plane can be swept through a volume to help form a mental three dimensional model. Helman and Hesselink used critical point analysis to extract the vector field topology and divide the flow into regions[HH90]. The investigator can then view a high level abstraction of the underlying flow data. Klassen and Harrington's *Shadowed Hedgehogs* used directed line segments set at surface grid points to visualize velocity[KH91]. A shadow was added to each line to encode orientation. Cabral used line integral convolution (LIC) to filter an input image of white noise and smear it along local stream lines in a plane[Cab93]. Other researchers have expanded on this technique to allow it to be applied to contour surfaces in a three-dimensional flow[MCG94, For94].

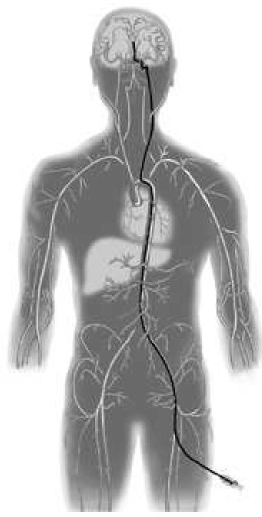

Figure 1.6: Path of microcatheter for endovascular brain aneurysm treatment.

Figure 1.7: Occlusion of intracranial aneurysm with platinum coil.

Two classical tools for visualizing a three-dimensional flow field are streamlines and particle traces. Other tools have been developed for visualizing special aspects of a vector field. Schroeder, Volpe, and Lorensen have developed *Stream Polygons* which are regular polygons with normals oriented to the local velocity vector[SVL91]. These polygons follow the flow and deform to reflect the local stress. Another tool called *Surface Particles* was developed by van Wijk who modeled flow particles as very small facets[vW92]. By controlling the emission of these particles, other flow visualization tools can be simulated, such as flow ribbons or tubes. Ma and Smith introduced a technique called *Virtual Smoke*[MS92]. Volume visualization techniques are used to simulate the movement of colored smoke in the field. *Streamballs* were developed by Brill et. al. as an extension to the particle trace technique[BDH+94]. Particles are modeled as globular metaballs that merge and separate as they move through the flow. Crawfis, Max, and Becker have introduced a construct called flow volumes[CMB94]. These are continuous three-dimensional semitransparent bodies that are carried along with the flow.

Stress fields within a three-dimensional fluid are mathematically represented as second order tensor fields. Such a tensor field has nine dependent variables at each point. This makes tensor fields difficult to visualize. Researchers are beginning to find ways of effectively performing this visualization[DH92, vWvL93, HM94]. Other investigators are exploring techniques for designing visualization interfaces that tie together the flow data, visualization tools, and user.

1.2.2 Design Methodologies

Besides constructing specific tools, investigators have developed methodologies for the design and selection of visualization tools. Turner et. al. have introduced a methodology for object-oriented design of scientific visualization environments using dynamic variables[TGB+90]. This technique allows parameters of various objects in the environment to be either directly manipulated or linked to other object parameters. Other researchers have investigated the use of object-oriented techniques in visualization[FH94, PA94]. Robertson has introduced a methodology for designing visualization tools[Rob90]. The technique attempts to formally match the characteristics of a visualization tool to both the underlying data and the aims of the investigator. Related work has been done to automate the design process using a rule based system[RT93, BF92a], and others have tried to model the desires and abilities of the user of a visualization system[DG94]. Researchers at Brown University have developed a technique and toolkit for designing three-dimensional widgets[CSH+92, ZHR+93]. These widgets are directly manipulable icons that can be controlled through direct manipulation of attached handles. While these efforts have addressed some of the concerns of three-dimensional visualization, little effort has gone into visualization interface design in immersive virtual environments.

1.3 Virtual Reality

Virtual reality is a computer interface technology that transports a user into their data so it can be directly experienced and manipulated. With virtual reality, distracting components of the interface that are not directly related to the user's tasks are minimized. These components can be divided into information that flows from the computer to the user, and vice versa. For data flowing to the user, examples of these distracting components would be the windowing system, and events occurring in the workspace. Examples of information flowing from a user to the computer are using a computer syntax, typing on a keyboard, and pushing a mouse each of which are usually only indirectly related to the user's task. A VR system attempts to reduce these distractions so as to increase focus on the task, make learning the system easier, and make using the system more natural and enjoyable.

Stereoscopic head mounted displays with body position sensors and gloves to measure gestures can be used to create an interactive virtual environment. Systems measure the position of the user's head and generate a wide view image of a landscape of graphical objects. Position and gesture of the user's hands can be used for manipulating virtual objects. Three-dimensional sounds can provide audio cues of object position, motion, and state.

Workstation based programs for flow visualization were introduced as early as 1980[EJ94]. These systems are very effective for visualizing two-dimensional static flows. As simulated flows become more complex, conventional monitors and mouse-based systems become less effective. Investigating three-dimensional and transient flows can be frustrating using a mouse and screen. This problem is compounded for users having limited experience with scientific visualization systems. Due to its three-dimensional and potentially intuitive interface, virtual reality can be an effective technology for dealing with these problems.

One application of virtual reality for visualization of numerically simulated fluid flows is the Virtual Wind tunnel[LB92, BL91, BF92b, BL92], developed at the NASA Ames Research center. Due to its immersive and interactive components, this system allows an engineer to rapidly explore a complex flow data set, using tools such as tufts (small pieces of attached string), streaklines (i.e. lines that would be generated by a dye injector), particle paths, and streamlines. The engineer may be testing a simulator, evaluating a design, or searching for a phenomenon.

Another application for virtual reality flow visualization is in hemodynamic simulation. One can imagine shrinking oneself down and swimming about in a blood vessel. The user could freely move to any area of interest within the flow. Hand gestures would allow different aspects of the flow to be highlighted. The three-dimensional nature of the blood flow could be seen naturally, and time could be completely controlled. The flow in an intracranial aneurysm has complex three-dimensional and transient behaviors[Har93]. Due to these features the visualization of intracranial aneurysm flow is an ideal candidate for a virtual reality system.

1.3.1 VR in Medicine

Researchers at University of North Carolina at Chapel Hill have been developing stereoptic displays, head mounted displays, and three-dimensional input devices. Two medical applications have been prototyped using this hardware. These applications allow physicians to comprehend and interact with complex three-dimensional medical structures. The first project involves an ultrasound image viewing system that superimposes the reconstructed image over the the real image of the patient[Fuc90]. Such a system allows a physician to effectively see through a patient's skin to the underlying anatomy. There are significant problems with processor speed and display technology that prevent this system from being practical.

A second system being developed at UNC is a virtual reality system for interactive radiation therapy planning[Fuc90, Rhe91, SBR+90]. The physician's task is to position a three-dimensional beam to expose a brain tumor to radiation. The path of the beam should be chosen to minimize damage to other areas of the brain. The application presents a three-dimensional model of the brain based on magnetic resonance data. Using a head mounted display and three-dimensional input device, the physician can explore the complex structure of the brain, locate the tumor, and manipulate a graphical representation of the radiation beam. It was found that physicians formed more specialized and nontraditional plans when using the interactive system, as opposed to the previously used batch system[RSS+91].

Researchers have built virtual reality systems that simulate surgery. Such systems are useful for the training physicians to perform delicate procedures. Some applications realistically simulate the interaction between anatomy and surgical tools[Sat93, Mer93]. Using a head mounted display, the surgeon in training can navigate the anatomy to learn its structure. Graphical scalpels and clamps can be used to operate on the simulated patient.

A laparoscopic simulator has been built that uses a workstation screen and a force feedback controller to maneuver a virtual probe[Sat93]. This system mimics the equipment and views used in actual laparoscopic surgery. The application is useful for teaching eye-hand coordination. A similar system is a lumbar puncture simulator that provides tactile feedback, but no visual feedback[BSCWW93]. Since it is used for training, this simulator rejects a more complex graphical interface to remain true to the procedure.

1.4 Aims of the Dissertation

The long term objective of this work is to reduce the danger of aneurysm surgery and to further knowledge of aneurysm hemodynamics. We propose to develop a computer system for endovascular therapy that aids in the simulation and visualization of blood flow structure. This system will help solve the problems of how to 1) visualize three dimensional vascular anatomy, 2) quantify flow, 3) determine the necessity of endovascular therapy, and 4) decide what particular therapy is appropriate.

Methods that have been used to gain insight into aneurysm hemodynamics are cineangiography, Doppler studies, and computer simulations. These methods have difficulties that preclude visualization of the details of the flow phenomena. Cineangiography is a commonly used technique. Its weaknesses as a visualization technique are that only bulk flow projections are visible. Since the images are two-dimensional, the three-dimensional nature of the flow and geometry must be mentally reconstructed from limited information. Endovascular Doppler measurements are difficult to use in vivo, and measure velocity ranges over a relatively large volume rather than at a specific point. While some computer simulations have been based on complex models (see Section 1.1.2), none have attempted to model clinically encountered aneurysms, validate those results, or provide a visualization environment for clinicians.

Our approach to these problems is to construct a system that will allow physicians to visualize complex anatomy and physiologic-based simulations. This system will: (1) acquire and model aneurysm geometry and other parameters using clinical case data, (2) simulate and validate transient three-dimensional blood flow by numerically solving the Navier-Stokes equations, and (3) visualize results using virtual reality and scientific visualization techniques.

A new methodology for interface design will be used to integrate all controls into the virtual environment, thus increasing immersivity and encouraging focus on the user's tasks. This will be accomplished by designing directly manipulable controls that will be integrated into the flow visualization tools. In order to minimize the interface complexity, navigation will be performed using voice commands.

It is hoped that this dissertation will address the following questions:

1. How can clinically encountered brain aneurysm geometry, blood flow, and simulated flow data be successfully modeled for subsequent use in a simulation/VR visualization system?

2. How useful are computational fluid dynamics techniques for application to solving pulsatile brain aneurysm blood flows?

3. How appropriate are virtual reality interfaces for visualization in a medical setting?

1.5 Test Problem

In order to test the approach described above, we have built a prototype VR visualization of simulated blood flow through a lateral wall aneurysm of a swine. The use of a swine model can be justified because swine physiology is considered close to that of the human. Using experimental animal models allows validation of models and simulations before application to clinical cases. By performing these preliminary simulations, real data will be available for testing the new methodology for design of medical VR visualization interfaces.

The aneurysm blood flow visualization system will utilize a wide array of hardware and software subsystems. These subsystems perform acquisition, archival, modeling, simulation and visualization roles. Figure 1.8 shows the processing stages that must be performed to convert DSA images into VR flow images. From the top down, the aneurysm images are processed to extract a surface description. This is converted into a mesh and input to the simulator. When a suitable solution has been found it is entered into an object-oriented flow database. The database is used by the VR system to continually update the objects in a scene. The dynamic virtual environment is created by looping from reading sensors to updating objects to rendering scenes.

1.6 Significance of the Dissertation

The risk of endovascular surgery for treatment of intracranial aneurysms has decreased, but is still too high to be ignored. One reason that the surgery remains dangerous is that surgeons have limited knowledge of the blood flow structure in aneurysms. This is true for aneurysms in general as well as specific aneurysms encountered in a particular clinical case. A computer system for simulating and visualizing aneurysm blood flow should help in determining appropriate therapy, and planning endovascular therapy. Such a system also provides a first step in understanding the probable outcome for individual patients, when combined with an outcomes database for populations of patients.

The justifications for this research are: the aneurysm blood flow visualization system should benefit the quality of medical treatment; and the techniques and insight developed in implementing the VR blood flow visualization system should be extendable to other areas of modeling, computer simulation, virtual reality, and scientific visualization.

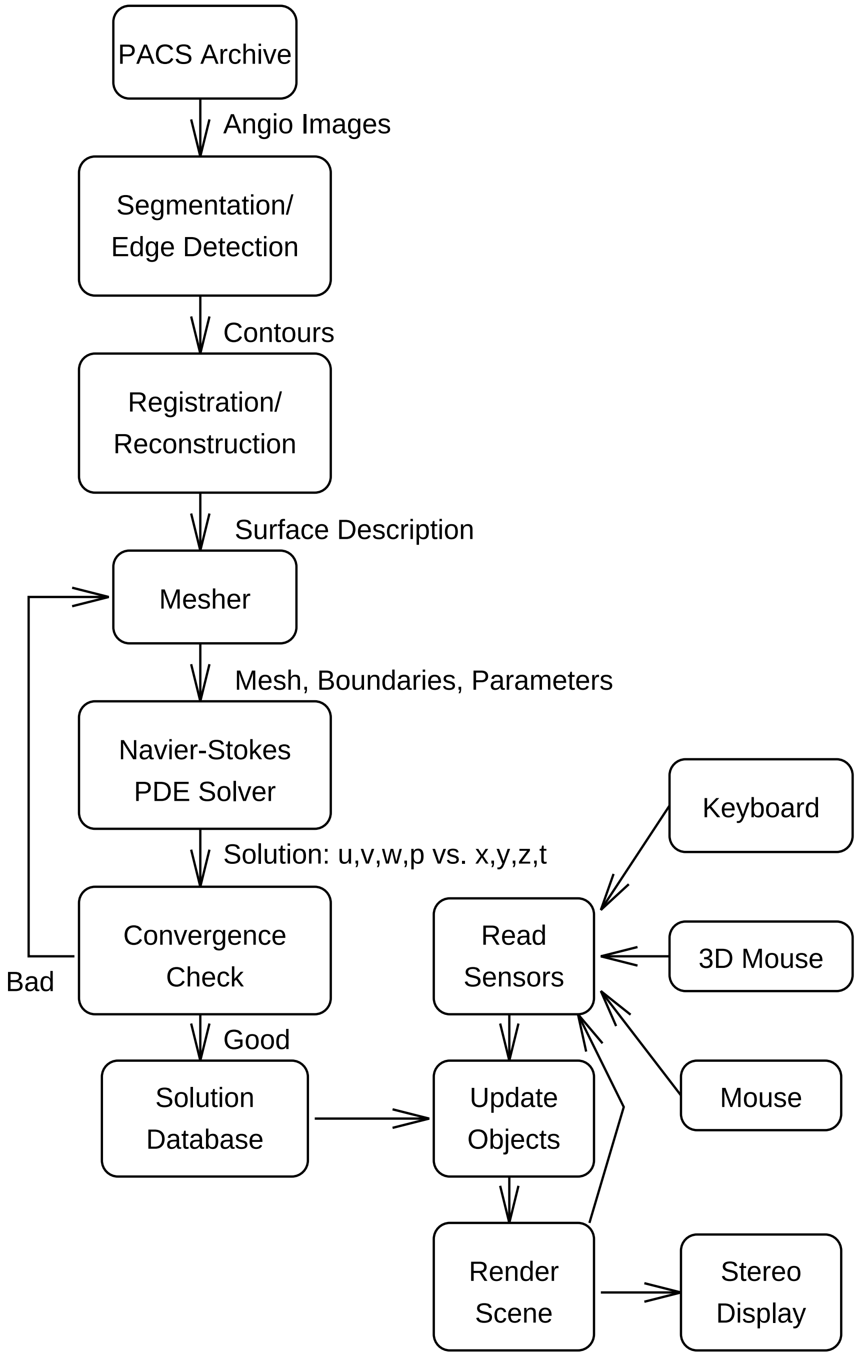

Figure 1.8: System overview displaying data flow through major components.

Chapter 2

Modeling

Modeling was done at different stages of the project to support the simulation and visualization of aneurysm blood flow. Before simulation, the physical geometry of the aneurysm and artery were modeled. Several strategies were used to build three dimensional meshes based on medical images. Then models of the physical blood flow were defined. Blood flow modeling involved general attributes such as transient fluid flow and blood properties as well as case specific properties such as blood pulsatility. Using simulation software to combine the geometric and physical models yielded solutions for transient velocity and pressure within the geometric model. In order to use this flow data interactively and rapidly in the virtual environment, an object-oriented data model was designed to organize and access flow information. This chapter describes these three modeling efforts.

2.1 Aneurysm Geometry

All aneurysm geometries were modeled based on sequences of 2D projected angiographic images. Angiographic images were acquired by x-raying the vasculature after endovascular injection of radio-opaque dye. The resultant image shows a projectional image of the dye. These images are used by physicians to mentally reconstruct the 3D geometry of the vasculature and to provide real time guidance of endovascular therapies. Figure 2.1 shows typical frames from an angiographic sequence in anteroposterior (AP) and lateral views. Image quality can be increased by subtracting the intensities of a pre-dye injection frame from a post injection frame. This process (called digital subtraction angiography or DSA) tends to remove unimportant structures from the image. Figure 2.2 shows the same images after subtraction.

The walls of the model were considered to be rigid. Though the walls in the swine aneurysm were seen to move slightly, it was impractical to model the elasticity in the first model. This simplification was justified by the small movements seen in the walls while viewing cineangiograms (angiogram movies) of the swine aneurysm. This assumption may be inappropriate if small changes in geometry are found to have large qualitative effects in blood flow. Such effects may be found in the neck region of certain aneurysms, where the flow bifurcates. It is known that the elastic properties of the vasculature have an effect on hemodynamics, as they adapt to conditions and help control vascular flow[Ste79].

Physically modeling and simulating vascular elasticity is a difficult task. An accurate model would have to include the representation of the vessel elasticity and constriction, as well as the forces exerted by the surrounding connective tissue. The vessel elasticity would vary depending on physiologic and geometric factors that are difficult or impossible to determine in vivo. An alternative and promising technique for modeling vascular elasticity is to use a changing geometry constructed from dynamic medical images. Rather than a free boundary, the movement of the boundary would be predefined. This method would bypass the need for a physical model, but require an imaging modality that is spatially accurate at relatively high frame rates.

The first step in modeling the geometry was to acquire AP and lateral angiographic sequences of the aneurysm. Particular images with high contrast were chosen from each sequence to represent the projection in that sequence. The initial image from each sequence was then subtracted from the representative image and the resultant image was normalized. From these images, the two orthogonal views were then registered

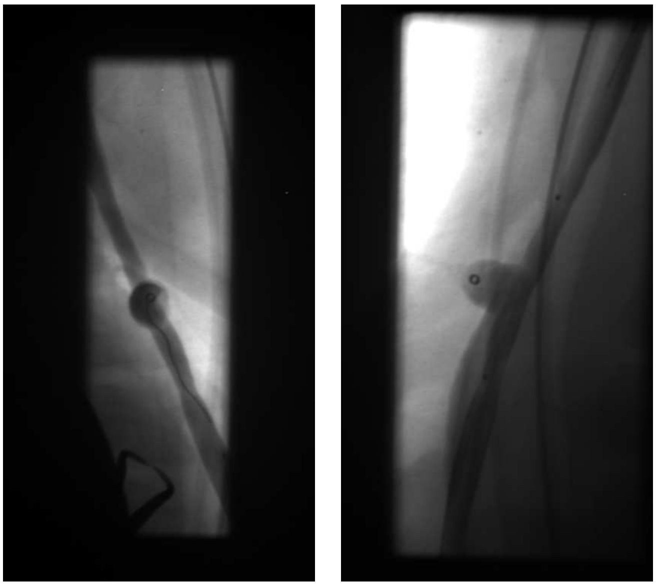

Figure 2.1: Anteroposterior and lateral angiographic projections of swine aneurysm.

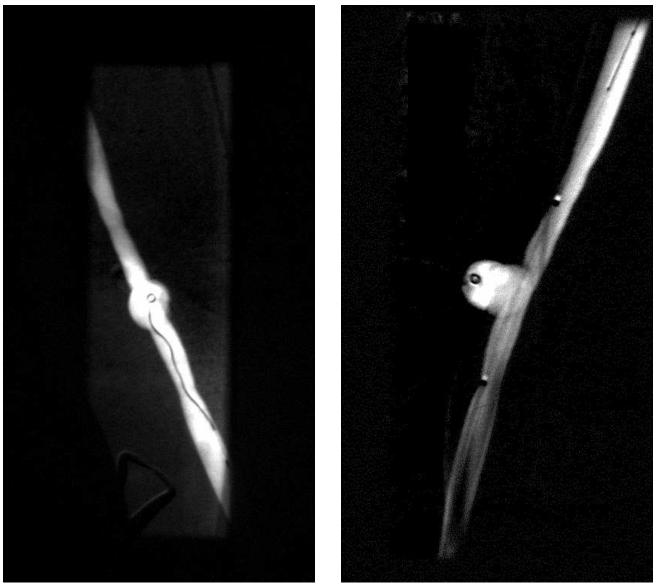

Figure 2.2: Subtracted anteroposterior and lateral angiographic projections of swine aneurysm.

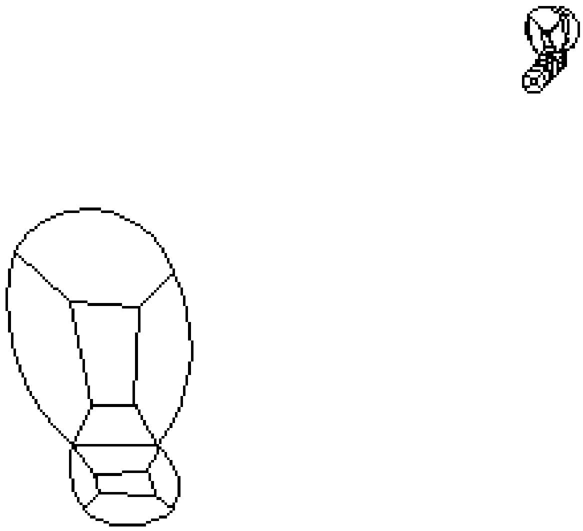

Figure 2.3: Quadrilaterals of one slice of typical hexahedral mesh. On the left, an aneurysm atop an artery. On the upper right, a small isometric view of the geometry

together into one scaled coordinate system by comparing scale markers and other geometric features of the images. Coordinate measurements were then made at points along the vascular wall in each view. These coordinates were then used to build the surface and interior mesh descriptions.

The final geometric models consisted of 3D descriptions of the surface as facets, and a conforming interior mesh. Two different techniques were used to model the aneurysm and artery geometry. Both techniques started with the same surface point coordinates.

2.1.1 Sliced Hexahedral Mesh

The first method generated a hexahedral mesh that was used in the spectral element method solver. The geometry was divided into a progression of slices normal to the artery axis. Slice thickness was varied depending on the geometry detail of the particular region. Where there were greater changes in the cross section dimensions, more slices were made. For each slice, a floor cross section was then drawn based on the surface point coordinates recorded in the first stage. Elliptic curves were manually fit through these coordinates for each cross section. The interior region of the floor of each slice was then filled with large quadrilaterals. The quadrilateral edges were curved when necessary to fit the cross sectional shape. The quadrilaterals conformed topologically to the shapes in neighboring slices. Hexahedral blocks were then built connecting the quadrilaterals in adjacent slices. Special care had to be taken to separate the sequences of aneurysm and artery hexahedra at the aneurysm neck. Figure 2.3 shows a typical mesh slice from a section within the aneurysm. In the lower left figure, an aneurysm cross-section is seen attached to an artery cross-section. Each cross-section is comprised of five quadrilaterals for a total of ten. Other similar slices (somewhat visible in the upper left isometric view) are defined separately. The commercial simulation preprocessor Prenek[1] was used to manually construct the hexahedral mesh.

Due to the large hexahedral blocks used to build the geometry of the model, the aneurysm wall contours were somewhat coarse. This was considered to be tolerable for early trials based on the assumption that sharp corners in the model would collect stagnant fluid and effectively round the walls at those points. Due to simulation sensitivity to neck configuration, more detailed geometry was later constructed (this is described in detail in Section 2.1.2).

In general, this technique for aneurysm mesh generation was found to be unsatisfactory. Manual meshing was found to be very time consuming. The slicewise generation requirement imposed by the preprocessor

[1] Fluent Inc.

was constraining, especially in the detailed neck region. Due to the hexahedral element shapes, and the requirement for a geometrically conforming mesh, refinements made in critical regions propagated to non-critical regions of the mesh. This increased the number of elements and computational load needlessly. This problem was especially critical in the neck region where both high and low flow gradients are found adjacent to each other. In general, modeling using large blocks also made it difficult to model the detailed geometry around the neck of the aneurysm. Since this meshing technique had many drawbacks, a more flexible irregular tetrahedral scheme was utilized.

2.1.2 Irregular Tetrahedral Mesh

The purpose of using an irregular tetrahedral mesh was to provide more flexibility in mesh generation. The fine details of the aneurysm neck surface could be modeled more easily. Elements can adaptively added to regions with high gradients without disturbing connected regions. Also, software tools are available for automatically filling space within a triangular surface mesh with tetrahedra. This reduces the problem to creating a surface mesh.

The general approach used to build the tetrahedral mesh was to construct a single cubic splined surface from the coordinate points, to fit a triangular mesh to that surface, and to automatically fill the internal space with tetrahedra. Surface modeling was done using the computer aided design (CAD) package Cadkey[2].

The first step was to register the projected coordinates into the same three dimensional system. The coordinate points (from Section 2.1) of the two projections were constructed in their respective projection planes based on manual measurements of coordinates from angiographic images. Figure 2.4 shows the points from the lateral projection. Splines were then fitted in those two planes for the four edges of the artery (two boundary curves per projection planes). Two center splines were then constructed that were half way between the top/bottom and left/right splines. The lateral splines with their center spline can be seen in Figure 2.5. The center splines were then extruded ninety degrees from their construction planes to form surfaces. An example of these two surfaces along with the four boundary splines is shown in Figure 2.6. The intersection of these two surfaces was the curve representing the centerline of the artery in three dimensional space. The surface points of each projection were then projected onto the center surface of the other projection. This shifted each projected surface point up or down to its correct position. The aneurysm surface points were then shifted up or down the same distance as their neighboring artery points. This was based on the assumption that the aneurysm generally is adjacent to and follows the artery. The points were then ready to be surfaced.

The next step was to build individual surfaces for the artery and aneurysm. For the artery, circumferential closed cubic splines were constructed at fixed distances along the axial flow curves. A cubic surface was then fitted using the axial and circumferential splines. For the aneurysm, two extra curves had to be constructed. The first was an “equator” curve that was constructed from the AP view points. This closed curve was positioned halfway along the axis of the aneurysm. Since there was no axial angiography projection made, some guess had to be made as to the aneurysm outline in that view. A spline was drawn for that view that conformed to the lateral projection, the “equator” curve and the artery surface. Control points were manually adjusted on this curve to smooth it out while retaining connection to the other curves. Finally, the aneurysm cubic surface was constructed by distorting a square patch to follow the other control curves. Figure 2.7 shows wireframe models of the individual aneurysm and artery surfaces. The next step was to construct a single smooth surface integrating these two unconnected surfaces.

Surface integration was performed by taking the axial center curve (the intersection of the extruded surfaces shown in Figure 2.6.) and projecting it outward onto the two surfaces shown in Figure 2.7. Eight projections were made to eight fragmented flow curves. Each of these fragmented curves was then smoothed into a single spline. Figure 2.8 displays these eight flow curves. Finally a cubic surface was fitted to these eight axial splines. Special consideration had to be made to concentrate nodal points in the artery under the aneurysm neck. This was because the surface area and detail was much greater in the aneurysm than the adjacent artery section. Figure 2.9 shows the completed integrated wire frame surface model.

To complete the surface model, end cap surfaces were constructed at either end of the artery. Next, a (triangular) polygon mesh was fitted to the surfaces. The software package TGrid[3] was then used to

[2] Cadkey Inc.
[3] Fluent Inc.

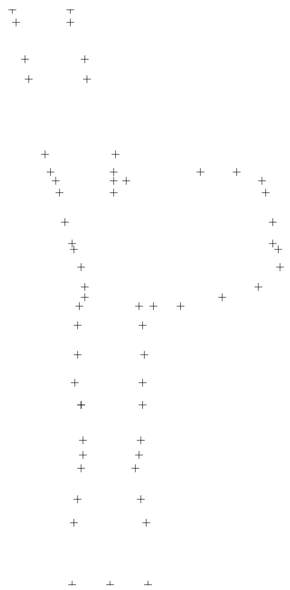

Figure 2.4: Lateral view projection points.

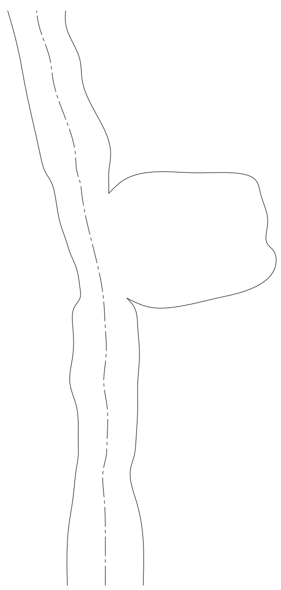

Figure 2.5: Lateral view splines with constructed center curve.

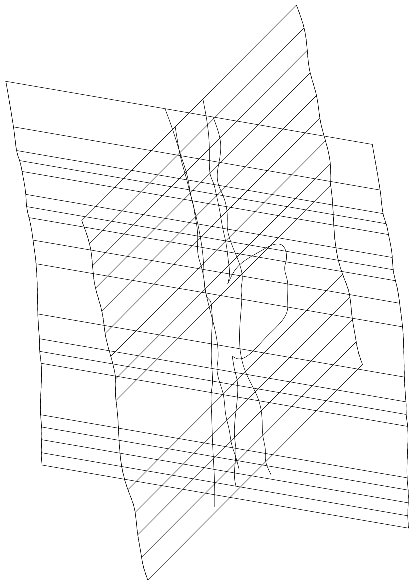

Figure 2.6: Center surfaces and original boundary curves before registration.

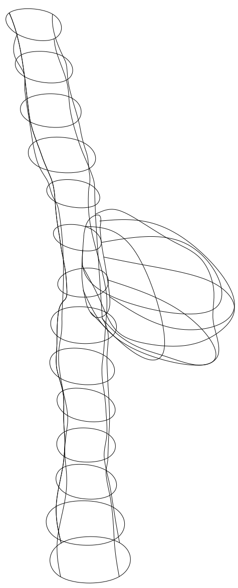

Figure 2.7: Original artery and aneurysm surfaces before integration.

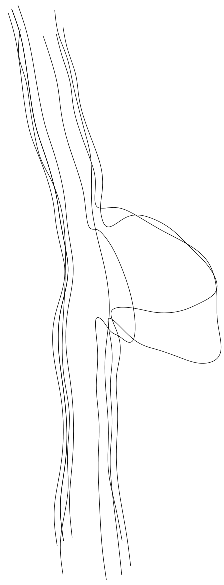

Figure 2.8: Axial flow splines.

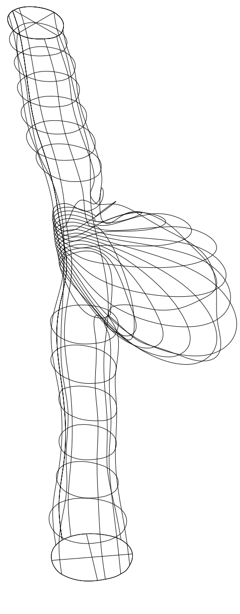

Figure 2.9: Wire frame surface model.

automatically construct a volume mesh within the surface mesh. The volume mesh could then be used as a geometric model for simulation.

2.2 Blood Flow

The physical system modeled was blood flow in an artery with an aneurysm. The aneurysm was located lateral to a relatively straight section of the common carotid artery. Blood flow pulsed following the systolic and diastolic portions of the cardiac cycle causing the aneurysm to deform slightly. The blood is known to have non-Newtonian viscosity, and the geometry was not symmetric.

2.2.1 Flow Regime

The flow was modeled as being laminar and incompressible, thus without turbulence. Turbulent conditions are when the fluid moves randomly in space and time. The no turbulence assumption was based on examination of the Reynolds number. Reynolds number is a non-dimensional parameter that measures the relative effects of viscous and inertial forces defined as[Bro67, RC85]:

$$Re = \frac{\rho u D}{\mu} \tag{2.1}$$

where ρ is density, u is local speed, D is a characteristic dimension, and μ is the fluid viscosity. It is often used as a predictor of the onset of turbulence. It is expected that when flow in a cylinder has a Reynolds number less than 2000, the flow will remain laminar[RC85]. Using conservative systolic values of: ρ = 1240 kg/m^3; u = 1.2 m/s; D = .004 m (arterial diameter); and μ = .005 kg/m·s, Re is calculated to be 1190, which is in the laminar range for Newtonian pipe flow.

Another consideration is that blood shows antiturbulence properties due to its make up. It has been found that thixotropic (shear thinning) fluids such as blood can be pumped at higher rates than Newtonian fluids with the same pressure drop. Also, small eddies tend to survive longer in thixotropic fluids, delaying the onset of turbulence. Taken together, these effects cause an increase of the transitional Reynolds number from 2000 to around 50000[Din85]. Thus turbulence should be rare in normal situations, while it may occur in pathological cases of low blood viscosity.

2.2.2 Governing Partial Differential Equations

The fluid is modeled as three-dimensional, laminar, and incompressible. The flow is governed by the laws of mass and momentum conservation. The Navier-Stokes equation describes momentum conservation in a viscous fluid. It can be written as a partial differential equation as[Bro67, Shi85]:

$$\rho(\partial_t \vec{u} + (\vec{u} \cdot \nabla)\vec{u}) = -\nabla p + \mu \nabla^2 \vec{u} \tag{2.2}$$

where ρ is fluid density, $\vec{u}$ is the velocity vector, p is pressure, and μ is viscosity. The ∇ symbol represents the gradient operator and ∂_t represents a partial derivative with respect to time.

The four terms in the Navier-Stokes equation represent balancing contributions to fluid momentum change in an infinitesimal fluid element. The term $\rho\partial_t \vec{u}$ represents the change of momentum with time in a differential element. The term $\rho(\vec{u} \cdot \nabla)\vec{u}$ corresponds to the momentum convected into the element by the flow current. The ∇p term represents change in momentum over time due to net pressure forces acting on an element. The $\mu\nabla^2\vec{u}$ term corresponds to the momentum change in time from friction induced shear forces.[Pat80] The Newtonian fluid model sets the viscosity μ as constant. In general, however, the viscosity is functionally dependent on the shear rate and shear history.

The conservation of mass requirement for incompressible flows can be enforced using the continuity equation[RC85, Shi85]:

$$\nabla \cdot \vec{u} = 0 \tag{2.3}$$

Thus the sum of inflows and outflows of an infinitesimal element is zero, and the fluid is incompressible.

2.2.3 Boundary Conditions

The boundary conditions for Equations 2.2 and 2.3 consist of both Dirichlet and Neumann boundary condition types. The Dirichlet or essential boundary conditions are used where only the velocity is specified on a boundary. In this model essential boundaries are the incoming arterial flow (velocity specified based on bulk flow measurements) and the rigid walls (velocity equal to zero, see below). The Neumann or natural boundary conditions occur where only traction (strain rate and pressure) is specified. Natural boundaries in this model occur across the arterial outflow zone (relative pressure set to zero).

Boundary conditions were specified along the arterial and aneurysmal surfaces, and along cuts through upstream and downstream sections of the artery. On the walls the velocity was considered to be zero. This essential boundary condition is based on the assumption that the no-slip condition holds and that the walls are rigid. The no-slip condition refers to the fact that for any real continuum fluid flow over a solid surface, the flow velocity is exactly zero at the surface[And84]. Thus any movement of flow at a solid surface is due to (and equal to) the movement of the surface itself. Since the walls are modeled as rigid, the total flow velocity at any point on the wall is zero. The natural boundary condition at the outflow was specified by assuming that there was a constant zero pressure in the outflow boundary surface. This condition is based on the assumption that the velocity profile has stabilized downstream of the aneurysm. All pressures calculated in the artery and aneurysm are considered to be relative to the outflow pressure.

The inflow boundary condition is more complex than the outflow condition. Ideally, the precise velocity profile across the surface at each point in time should be given. No hemodynamic measurements were available that would determine this velocity profile. The solution to this problem was to move the input boundary farther upstream, and model the input velocity as a simpler profile while still maintaining the proper mass flow rate. The actual input velocity profile was then modeled as a transient plug flow. As long as the boundary was far enough away from the aneurysm, the deviant velocity profiles due to inexact boundary conditions were resolved before the flow reached the aneurysm.

Though constant across the inlet section, blood flow was varied through time. In order to relate the maximum arterial flow (at any time) to the constant value of the real inflow, assumptions had to be made on the real flow profile. Steady state Newtonian pipe flow has a parabolic profile, which was chosen as an initial model. The axial velocity is thus defined as:

$$v(r) = V_{max}\left(1 - \left(\frac{r}{R}\right)^2\right) \tag{2.4}$$

where v is axial velocity at a point, r is the radius of the point V_{max} is the maximum velocity (i.e. reading from the Doppler probe), and R is the radius of the pipe. Since we are trying to model conservation of bulk flow, we need to find the mass flow rate through the pipe. This can be calculated by integrating the velocity in Equation 2.4 across the pipe:

$$\dot{m} = 4\rho \int_0^{\frac{\pi}{2}} \int_0^R v(r) r dr d\theta \tag{2.5}$$

which yields a solution of:

$$\dot{m}_{parabolic} = \frac{1}{2} V_{max} \rho A \tag{2.6}$$

where A is the cross-sectional area of the pipe ($= \pi R^2$.)

For plug flow the axial velocity is constant and is expressed:

$$v(r) = V_{max} \tag{2.7}$$

Integrating Equation 2.5 with Equation 2.7 yields the mass flow rate for plug flow:

$$\dot{m}_{plug} = V_{max} \rho A \tag{2.8}$$

It can be seen then that for a certain mass flow rate, cross-sectional area, and density, maximum velocity in parabolic flow is twice that of plug flow. To maintain mass flow rate, the inlet blood speed at any particular time in the heartbeat cycle was thus set at half of the respective Doppler measured maximum velocity. A lead length in the artery was provided to allow the flow to develop to its roughly parabolic profile before reaching the aneurysm.

2.2.4 Initial Conditions

The initial conditions provide a starting point for a transient solution. They consist of a starting velocity field within the system boundaries. From initial trials it was found that the effects of initial conditions mostly died out after about 0.05 s of simulated time. Since the system seemed to respond quickly to transients it was convenient and tolerable to start with a zero velocity field. The simulation was typically run through one and one half heartbeat cycles, so that the initial half cycle (typically 0.5 s) of data could be replaced with the first half of the second cycle. Using this technique, the effects of an inaccurate initial condition were mostly eliminated. This was verified by comparing the values for the first half of the first and second cycles.

2.2.5 Blood properties

The two physical properties most important for modeling fluids are density and viscosity. The density of blood was modeled as a constant value of 1240 $\frac{kg}{m^3}$ in all simulations. Blood viscosity is much more complex, and four different models were devised and compared.

The dynamic viscosity (μ) of a fluid is the ratio of the shear stress (τ) to the strain rate ($\dot{\gamma}$.) The viscosity equation is written:

$$\tau = \mu\dot{\gamma} \tag{2.9}$$

The shear stress is the force per area that is applied to move one laminar fluid plane over another. The strain rate is the velocity difference of the planes divided by the distance between them. Viscosity is thus the scaling factor for the frictional force that resists shearing in a fluid.

Many fluids (such as air and water) exhibit a linear relation between shear stress and strain rate. These fluids have a constant viscosity and are called Newtonian. Other fluids (called non-Newtonian) have a nonlinear relation between shear stress and strain rate. Rather than being a constant parameter, viscosity in non-Newtonian fluids is a function of the strain rate. For non-Newtonian fluids, the viscosity equation can be written as: $\tau = \mu(\dot{\gamma})\dot{\gamma}$. [Boh87]

Human blood is known to be a non-Newtonian fluid. Due to aggregation of blood cells in the plasma, and effects of other materials, blood exhibits shear thinning behavior. This and other thixotropic fluids have decreasing viscosity as shear rate increases. A mathematical model for blood viscosity (defining $\mu(\dot{\gamma})$) was formulated based on observations made in the literature on the viscous properties of human blood. The major factors considered were:

High shear. At shear rates above about 100 s^{-1}, blood asymptotically approaches Newtonian behavior with a constant viscosity dependent on various blood parameters such as haematocrit and fibrinogen levels. Typical high shear viscosity values reported for normal human blood is in the range 3 – 6 centipoise.

Low shear. At shear rates below 100 s^{-1}, blood viscosity increases generally following a power law model. A Casson model is often used in this range which relates viscosity to the inverse square root of shear rate[Mil89, Din85].

Very low shear. At sufficiently low shear rates the viscosity of blood becomes effectively infinite and blood becomes solid. This is when blood clots are formed. This phenomenon is not instantaneous, but typically takes minutes to occur.

Four models of blood viscosity were formulated and simulated for comparison. The first two models used were simple Newtonian models, using viscosity values of 5 and 40 centipoise. The first viscosity (5 centipoise) comes from a typical high shear (¿ 100 s^{-1}) value for human blood. The 40 centipoise value for viscosity is typical for low shear rates ($\approx$ 2 s^{-1}).

The third and fourth models were non-Newtonian. These two models differ only by the low shear cutoff for shear thickening. The mathematical models for non-Newtonian viscosity were (with μ in centipoise and $\dot{\gamma}$ in s^{-1}):

$$\mu = \begin{cases} 200 & \dot{\gamma} < 0.1 \\ 58.5\dot{\gamma}^{-0.534} & 0.1 \leq \dot{\gamma} < 100 \\ 5 & \dot{\gamma} \geq 100 \end{cases} \tag{2.10}$$

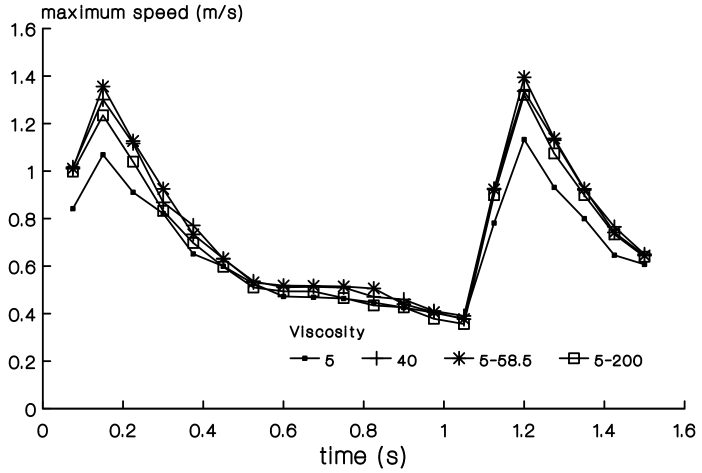

Figure 2.10: Comparison of simulated maximum blood speed for different viscosity models. Each point is a global maximum for the corresponding model, thus the positions are not necessarily equivalent. Data is from swine 895.

and

$$\mu = \begin{cases} 58.5 & \dot\gamma < 1.0 \\ 58.5\dot\gamma^{-0.534} & 1.0 \leq \dot\gamma < 100 \\ 5 & \dot\gamma \geq 100 \end{cases} \tag{2.11}$$

For $\dot\gamma \geq 100$ (the Newtonian range) the value of 5 centipoise was chosen from the literature as a typical high strain rate viscosity value. In the low strain rate range ($0.1 \leq \dot\gamma < 100$) a power law equation was fitted to experimental data from [Din85]. A double log plot of viscosity versus shear rate shows a roughly straight line from ($\dot\gamma/cP, \mu/s^{-1}$) points $(0.1, 200)$ to $(100, 5)$. Solving the power law equation over this range yields $\mu = 58.5\dot\gamma^{-0.534}$. This result is close in form to the Casson model which uses an exponent of -0.5. Since the modeled time (one heartbeat) is too short for thrombosis to occur, thrombus formation was not modeled. To stop the viscosity model from going to infinity in regions of zero strain rate, a maximum value had to be set on viscosity. Two limits were chosen as 200 cP for all $\dot\gamma < 0.1$, and 58.5 for $\dot\gamma < 1.0$. These boundaries had to be chosen somewhat arbitrarily, since little is known about the dynamic response of blood viscosity.

In order to compare the viscosity models, simulation trials were made on a single swine aneurysm model (swine 895) with viscosity the only varied parameter. The boundary conditions were pulsatile, the simulation was run for one and a half heartbeats.

Global values were calculated for each data set of maximum speed, maximum strain rate, maximum surface strain rate, and pressure drop. These values for the four viscosity models were then plotted on the same time axis. Figure 2.10 shows the maximum blood speed for each model. The maximum speed is driven by the blood in the center of the parent artery. The two non-Newtonian models and the high viscosity Newtonian model have very similar results, while the standard Newtonian model ($\mu = 5$ cp) yields approximately 20% lower speeds at systole. It is interesting to compare Figure 2.10 to Figure 2.11 which is the Doppler measured maximum arterial blood speed for the swine. This chart was the original source for the simulated inflow conditions. A parabolic inflow profile was used to convert the maximum blood speed into bulk flow. All four viscosity models yielded peak arterial values within 20% of the experimental measurements. This fact provides some validation of the arterial profile model used.

Figures 2.12 and 2.13 compare the global maxima of strain rate and surface strain rate between the four viscosity models. It can be seen that there is a significant variation across viscosity models in the global strain rate, less in surface strain rate. It appears that the maximum strain rate is governed by the maximum viscosity allowed by the model. The models with higher allowed viscosity had lower strain rates. This could

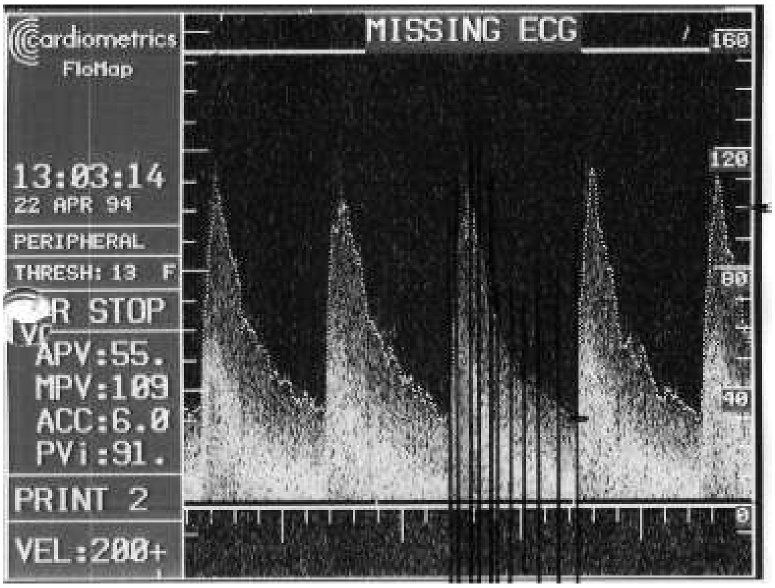

Figure 2.11: Doppler probe maximum blood speed for swine 895.

be expected by examining the viscosity law at a given stress.

As a second approach to comparing viscosity models, maximum pointwise differences were computed and charted for velocity and strain rate. In this way, local differences could be analyzed. The two viscosity models that showed greatest global differences were compared, namely, the 5 cp Newtonian and the 5-200 cp non-Newtonian models. The plots show the values of the non-Newtonian model subtracted from the Newtonian model. Figure 2.14 shows the local maximum differences of speed versus time. The Newtonian simulation had values at systole 60 cm/s greater than the non-Newtonian one.

Figures 2.15 and 2.16 show maximum local differences in strain rate and surface strain rate respectively. Again significant differences appear between the Newtonian and non-Newtonian models when a local comparison is made..

These comparisons indicate that there is a significant difference in the results depending on whether a non-Newtonian model is used.

2.3 Flow Data

The output format of the CFD simulation packages used was a simple flat file description of nodal values of velocity and pressure at discrete times. In order to visualize this data interactively, the access and computation speeds had to be increased considerably. During each visualization frame, the input devices had to be read, and a new image had to be constructed. In order to form this image, the appropriate data had to be extracted from the flow data base. Using commercial visualization packages it was found that flat files were incompatible with interactive real time visualization. There were four main performance problems found with using the flat file format for flow data:

1. Locating the correct data corresponding to a particular position and time.
2. Performing spatial and temporal interpolation.
3. Traversing into adjacent areas of the flow.
4. Handling large numbers of queries per timestep.

To address these problems, the flow data was remodeled into an object-oriented design. The four problems above were addressed in the design by:

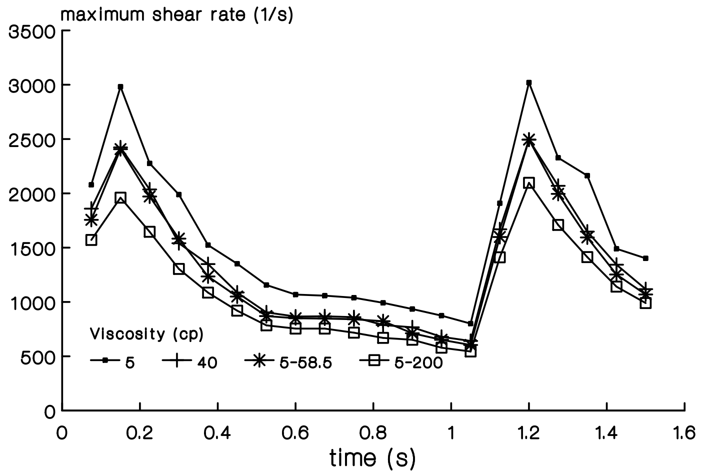

Figure 2.12: Simulated maximum blood strain rate for different viscosity models. Data is from swine 895.

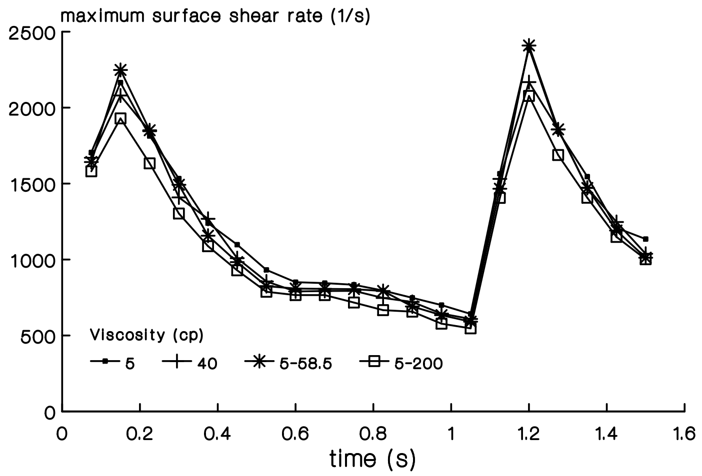

Figure 2.13: Simulated maximum surface strain rate for different viscosity models. Data is from swine 895.

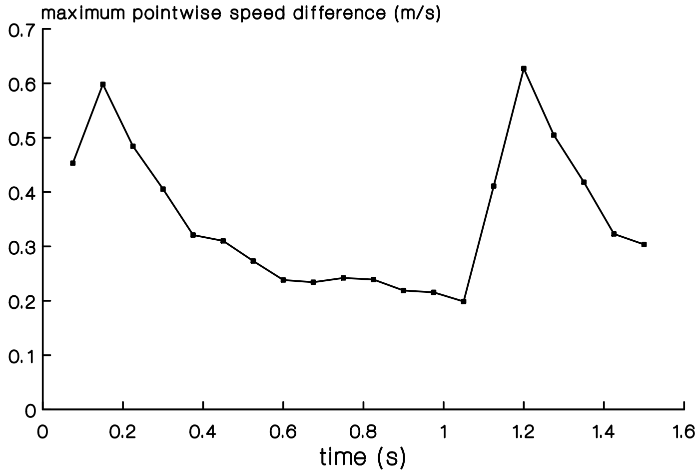

Figure 2.14: Simulated maximum pointwise difference of speed between Newtonian and non-Newtonian viscosity models. Data is from swine 895.

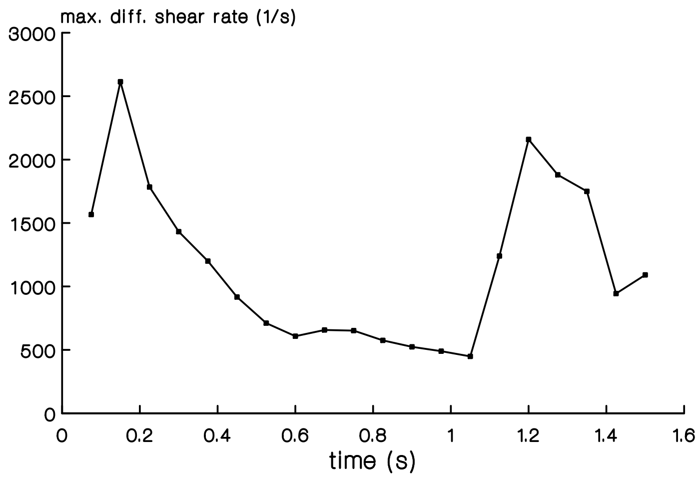

Figure 2.15: Simulated maximum pointwise difference of strain rate between Newtonian and non-Newtonian viscosity models. Data is from swine 895.

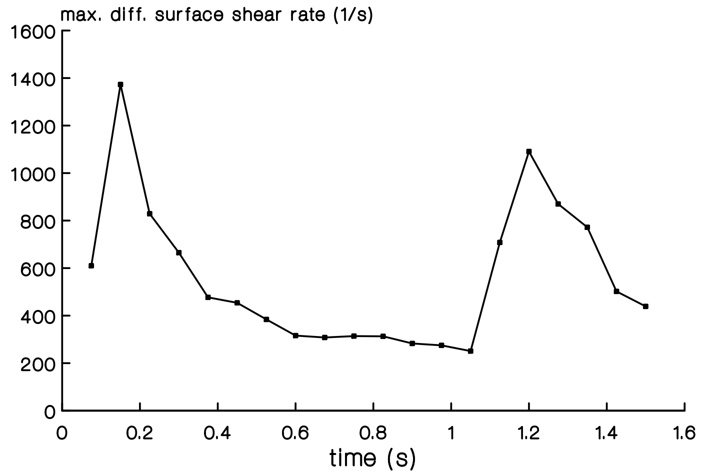

Figure 2.16: Simulated maximum pointwise difference of surface strain rate between Newtonian and non-Newtonian viscosity models. Data is from swine 895.

1. Overlaying a fast cubic spatial index over a tetrahedral mesh.
2. Precomputing linear field equations for each tetrahedron.
3. Building an embedded adjacency map into the objects.
4. Enabling parallel threaded queries for symmetric multiprocessors.

The output from the simulation is velocity and pressure values at many nodal points within each element and at discrete times of the heartbeat. The Nekton package uses a spectral element method which returns a high order solution over each large element within the domain. The Fluent unstructured packages outputs flow data linearly distributed over an irregular tetrahedral mesh. This flow information is entered into an object oriented data base for access by the visualization program. The data base preprocesses the raw flow data to provide fast indexing, spatial and temporal interpolation, and computation of secondary variables. The flow data base organizes the data into tetrahedral cells that contain linear fields of the primitive data. Cells are placed with other objects (e.g. nodes, faces) within an object oriented hierarchy that comprises a complete transient flow field. By converting single large high order hexahedral elements into many low order tetrahedral cells, many advantages are gained [KN91]. Determining whether a position is within a cell is easier for a simple tetrahedron than a curved and twisted hexahedron. A linear solution over a tetrahedron can be presolved using a simple linear matrix inversion - for a hexahedron, the solution is generally nonlinear. Tetrahedra are a more primitive geometric entity that can be used with other CFD package element configurations.

2.3.1 Class Design

Following is a description of the classes involved in the Flowfield package. Together they model a continuous transient and three dimensional flow field within a region. This is performed by discretizing the field. The space is cut up into tetrahedra each of which contains a linear field of data. The time is cut up into steps. The Flowfield can interpolate a data point given an arbitrary position and time. Locality of position can be exploited if necessary. The complete source code for the Flowfield class library is available as an appendix to a UCLA Computer Science Department 1996 technical report.

The interpolation performance is increased by precomputing during object construction. The field matrix for each cell is inverted to compute the linear equation for velocity and pressure. The plane equation for each face is precomputed to allow quick determination of whether a position is within a cell. A spatial index is constructed to provide likely cells in which an arbitrary position might be. Cell adjacency information is computed and stored to allow the space to be rapidly traversed in search of a local cell.

The classes were chosen to encapsulate the concepts and objects in a meshed flow field. Tetrahedral cells were chosen because most meshing geometries can be easily converted into this geometry through tetrahedronation (e.g. hexahedral Nekton geometry are readable.) Each class is responsible for its own input and output, using streams. The input and output stream operators for each class are symmetric. This organization defines a complete file format (called a .ff file) to be used that contains all information necessary for rebuilding the Flowfield object. Thus once the object is constructed and written out to its native format, it can be reconstructed later without going through the time consuming precomputation process.

It is quite common for one of the more primitive objects to be used many times by one of the higher level objects. For example, one node is shared by many cells. These redundancies were eliminated to reduce data size and coherency problems. At the top level (Flowfield), the primitives are organized into tables. At the lower levels, they are referred to by indices into those tables.

The main interface to the Flowfield library is the Flowfield class. After initialization, the main interface used in this class is the eval member function. This function takes a position/timestep as an argument and returns a Flowpoint. From the Flowpoint, the library user can extract pressure, velocity, maximum speed, speed in a direction, maximum strain, or strain on a plane. The eval function can alternatively accept an array of position/timesteps and return a corresponding array of Flowpoints. This threaded version of the function divides the computational work equally among the processors. The processors work independently, sharing memory. After all work is completed, the array of all results is returned. The performance of the parallel versus serial eval functions was tested on a four processor SPARC station. An approximately 3 times speed up was measured using four processors. It was determined by performing system level profiling that the efficiency loss to 75% was due to bus contention from multiple processors accessing shared memory.

The classes, generally from bottom to top, follow.

Position

Represents the spatial location of a point. Performs vector algebra operations.

Quaternion

Represents spatial orientation. Responsible for rotational algebra, rotating points, and inverse rotation. Depends on Position class.

Flowpoint

Represents the velocity and pressure of a fluid at one position and time. Computes maximum speed, and speed in a direction. Performs vector algebra and scaling. Depends on the Quaternion class.

Node

Represents flow at one point over a series of times. Control Flowpoints at a mesh point. Depends on Position and Flowpoint.

Face

Represents a connected triangle in space with an inside and outside. Handles traversal to an adjacent cell, reporting of external faces, automatic connection to other faces, determines if a position is inside or outside a cell. Depends on Position, and Node.

Linearfield

Represents a single linearly varying field of flow parameters. Automatically constructs field from four points, interpolates data at a point. Depends on Position, and Flowpoint.

Cell

Represents flow within a tetrahedron over continuous space and time. Responsible for constructing fields for each timestep, managing local adjacencies in the mesh; determining if positions are inside itself, and interpolating data at a given position and time. Dependent on Position, Flowpoint, Node, Face, and Linearfield.

Spatialindex

Returns a list of cells near a position. Constructs and interfaces a three dimensional index of cells. Depends on Position, and Cell.

Flowfield

Represents fluid flow in continuous space and time over a three dimensional region. Construct and organizes cells and nodes in tables. Reads in and digests Nekton data files. Performs global and local spatial/temporal interpolation of flow data. Depends on Position, Flowpoint, Node, Cell, and Spatialindex classes.

Figure 2.17 summarizes the relationships between the classes using a simplified Booch notation. The vector with a solid dot represents the has-a relationship. The side with the dot has one or more of the objects on the other side of the ray.

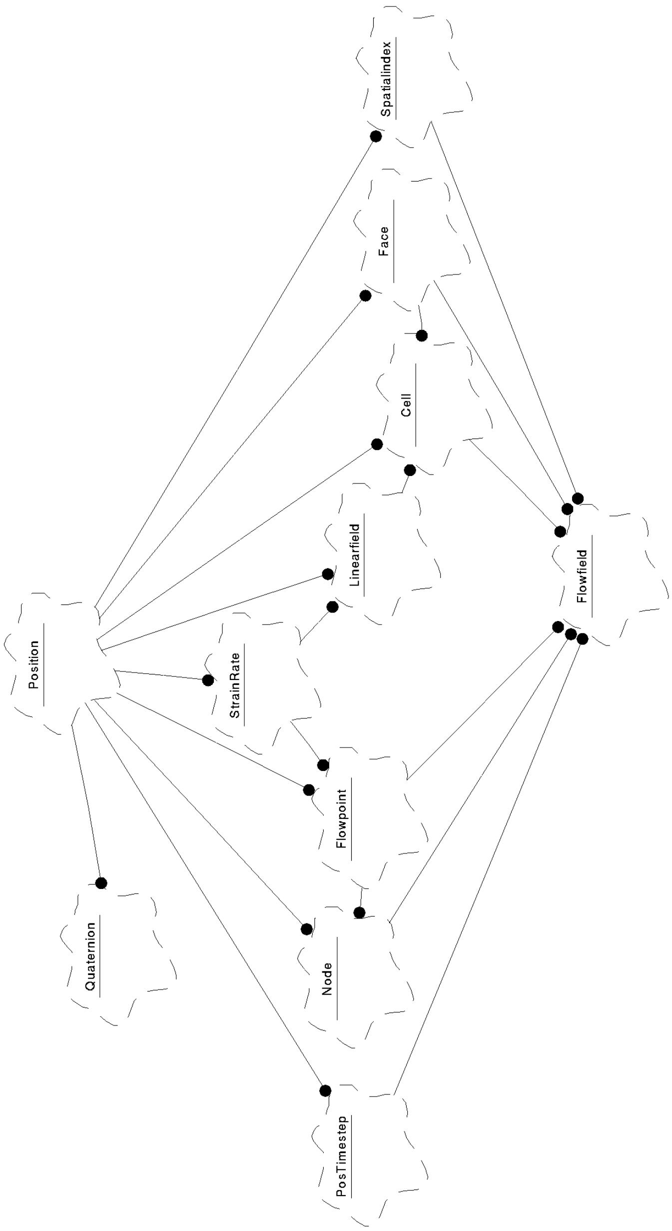

Figure 2.17: Flow field class library diagram.

Chapter 3

Simulation

Once the governing equations with boundary and initial conditions have been prescribed, the problem is ready to be solved. Since an analytical solution is impossible in most complex systems, the problem must be converted into a form that can be solved approximately on a digital computer. The initial method chosen for solving the Navier-Stokes and continuity equations was the spectral element method (SEM) which is an extension of the more general method of weighted residuals (MWR). This section presents an overview of the method of weighted residuals, a description of the SEM, consideration of other methods, and discussion of commercial software available for computational fluid dynamics (CFD). Finally, a validation study of the preliminary numeric results is presented and discussed.

3.1 Method of Weighted Residuals

One technique for solving partial differential equations such as the Navier-Stokes equations is to use the method of weighted residuals. The MWR is also known as the Galerkin method[Sch88]. The generalized algorithm corresponding to MWR can be stated in the following form:

1. Construct an approximate solution with unknown parameters
2. Insert this approximation into the differential equation
3. Define the residual to be the difference between approximate and true solution
4. Minimize residual in a weighted sense by:
    - 4.1. Multiplying residual by weighting function
    - 4.2. Integrating the product over the domain of interest
    - 4.3. Setting the result to zero
5. Apply Step 4 using a succession of orthogonal weighting functions
6. Solve for the parameters in the linear algebraic system generated in Step 5
7. Insert the calculated parameters into the approximate solution

The approximate solution constructed in Step 1 is comprised of a set of basis functions (also called shape functions). One method of defining the basis functions is to have them cover the domain in a piecewise continuous fashion. In this way each shape function corresponds to a particular zone in the domain. Another way of defining basis functions is to have each function defined over the entire domain.

The integration in Step 4.2 is performed by first rewriting the integral in the "weak" form. Using integration by parts, the order of the differential equation is lowered by one. This is accomplished by switching differentiation between the weighting function and the dependent variables. The resulting integral can be used to solve for the unknown parameters.[Sch88, Flub, Red84]

The MWR can be varied by choosing different basis and weighting functions. The method of weighted residuals encompasses the finite element, spectral, spectral element, finite volume, and boundary element methods. These five methods were evaluated for suitability to the flow regimes expected, and availability of corresponding computational fluid dynamics packages.

3.1.1 Finite Element Method

The finite element method is a common sub class of the MWR. The method was first used for structural analysis, but has become well established in computational fluid dynamics. In the FEM, the domain is divided into distinct elements. The basis functions are then defined with a one to one correspondence to the elements. Each basis function describes how the solution varies over the respective element. Nodal points are defined as the positions in an element where the solution will have the value of the corresponding parameter. The basis function for one element is defined to be unity at the element nodal point and zero at all other nodal points. The total solution is calculated by summing the products of basis function and corresponding parameters. The basis functions are usually defined as linear or quadratic. The parameters when solved represent the nodal values of the approximate solution over the domain.[Flub, Red84]

The FEM is especially suitable for problems with complex geometries. The method is often used with an unstructured mesh that can be more easily used to conform to nonorthogonal boundaries. It is simple to refine mesh spacing in critical areas. The FEM is generally quite accurate. There are some weaknesses associated with the finite element method, such as high computation time and memory requirements. The setup work for the FEM is quite universal, i.e. changing a problem parameter will not force a new setup.

3.1.2 Spectral Method

The spectral method is another member of the MWR family. In the spectral method, each basis function is defined globally. Unlike the FEM, each nodal point in the domain has contributions from all basis functions. Common choices for basis functions in the spectral method include Chebyshev and Legendre polynomials. In simple problems the spectral method yields rapid convergence rates. Unfortunately the spectral method has limited usefulness since it is difficult to use with nontrivial geometries.[Flub] Since the geometry of the artery and aneurysm are nontrivial, the spectral method was considered inappropriate.

3.1.3 Spectral Element Method

The spectral element method is an extension to the MWR that combines some of the advantages of the finite element and spectral methods. In this method, the geometry is partitioned into large elements. Since the domain is split into elements, more general complex geometries can be used, as in the FEM. High-order basis functions are defined over each element. This allows rapid convergence of simple fields, as with the spectral method. In this way the SEM overcomes the geometric complexity limitations of the spectral method while taking advantage of its rapid convergence. The spectral element method is advantageous when the solution is relatively smooth and the geometric domain is nontrivial.

In one implementation of the SEM[Flub], the space domain is divided into hexahedral elements. On each element, a three-dimensional Cartesian mesh of collocation points is constructed. Then the dependent variables are expanded over the collocation points as Lagrangian interpolants. These interpolants act as a basis for the solution of the governing equations . The basis is then inserted into the governing equations, and the Galerkin method is used to attain the corresponding algebraic equations. The Galerkin technique is an algorithm for applying the MWR. Convergence of the spatial solution is attained by increasing the number of elements in the domain or the number of collocation points per element.[Flub]

Based on the assumption that flow within the aneurysm would be smooth, the SEM was originally chosen for solving the Navier-Stokes and continuity equations in the aneurysm model.

3.1.4 Finite Volume Method

The finite volume method (FVM) discretizes a fluid field as an orthogonal grid (e.g. Cartesian or polar). A set of algebraic equations is generated by approximating the partial differential equations with appropriate finite difference equations local to each node. The FVM is also known as the finite difference method, the method of collocation, and the subdomain method. At its root, the FVM is another variation of the MWR, though its implementation is simpler than most of the other methods. The finite volume method has a successful history of use on computational fluid dynamic (CFD) problems.[Pat80, Flub]

Coding FVM systems is easier than the other methods, and the programs tend to have acceptable performance. One significant drawback is the orthonormal grid imposed by the method. This requirement makes complex geometries difficult to represent. Some commercial FVM packages have sought to overcome this limitation by introducing body fitted coordinate systems. Another approach to overcome the geometric restriction by hybridizing the FVM with the FEM. This approach leads to the Control Volume Finite Element Method (CVFEM) [Flua] used in Fluent/UNS.

3.1.5 Boundary Element Method

The boundary element method (BEM) is based on Green's Theorem. The method involves integrating to find the interior values of an element based on the function along its surface. It is difficult to simply introduce new conditions into a problem since this fundamentally alters the mathematics involved. The method only yields a solution on the element boundaries, and does not supply interior values. It is a highly mathematical method and requires significant initial preparation.

Problems best suited to the BEM are those with high gradients and infinite boundaries. Most applications have been in turbulent and sonic flow. The method yields good convergence rates, but has been found to be difficult to implement due to the mathematical expertise required during programming.[BM90]

3.2 Comparison of SEM and FEM

The spectral element and finite element methods belong to the general family of weighted residual methods. The FEM uses a piecewise linear or quadratic function for forming a basis over many discrete elements. The SEM uses a more complex basis consisting of higher order polynomials defined over typically larger elements. The solution to a SEM is then a smooth interpolation of collocation points. Figure 3.1 illustrates the difference in solution between the two methods.

The advantage of the SEM is that it provides fast convergence when solving smooth fields. The speed is due to the assumption of smoothness inherent in the method. The disadvantage of the SEM is that a field with high gradients will force small elements to be used that will compromise the speed gains. The FEM with its typically finer meshes is more suitable for fields with high gradients. Large changes can occur across grid points without disrupting adjacent calculations.

For low Reynolds number problems, the spectral element method takes advantage of the smooth transitions in the flow to yield faster convergence times than the FEM. Since the grid elements are larger, manual grid construction is faster, but constructing fine detailed meshes is more difficult.

3.3 Computational Fluid Dynamics Software

There are a variety of commercial CFD packages available. They vary in types of flows simulated (e.g. laminar, transient, compressible), numerical methods used, and user interfaces (text or graphics based). Several commercial CFD packages were considered for aneurysm blood flow simulation. They consisted of software packages implementing the finite element method, the finite volume method, and the spectral element method. The packages were compared based on the following criteria:

- Ease of grid generation
- User interface
- Relative performance
- Accuracy
- Availability and support

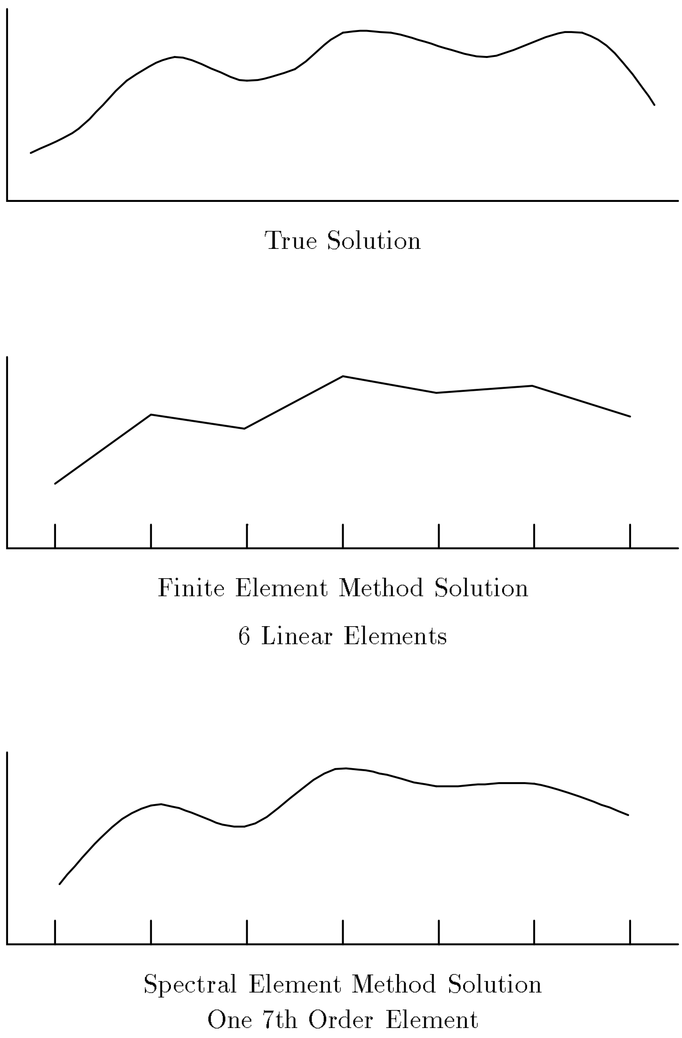

Figure 3.1: Illustration of FEM and SEM solutions.

- Visualization capabilities

Considered CFD packages included:

EasyFlow FVM package for general flow[1]

Fidap FEM package for general flow[2]

Flow3D FVM package with unstructured mesh[3]

Fluent FVM package with body fitting mesh for general flow[4]

Fluent/UNS CVFEM package with unstructured mesh for general flow[5]

Nekton SEM package for laminar flow[6]

P/Flotran FEM package for general flow[7]

Phoenics FVM package for general flow[8]

Rampant FEM package for compressible flows[9]

From these packages, the SEM code *Nekton* was chosen based on the assumption that velocity gradients in the aneurysm blood flow would be low.

3.4 Nekton

In order to solve a flow system using the SEM (Nekton), the problem had to be defined in a specific way. A geometric grid of hexahedral elements was constructed. The density and viscosity of the fluid had to be defined. Viscosity was defined as either constant or as a function of flow parameters at a particular point. Initial conditions were specified for each point in the space. Boundary conditions were specified on the outer faces of the elements for each instant of time. The order of the spectral elements was declared. Finally, a solution strategy had to be chosen.

In order to decrease solution time, the method of characteristics was used to solve the time dependent Navier-Stokes equation. Using this method convective and diffusion terms of the Navier-Stokes equation were solved using different time steps. The convective term, $\rho(\vec{u}\cdot\nabla)\vec{u}$, was treated explicitly. A third-order Adams-Bashforth multistep method was used yielding temporal accuracy of Δt^3, where Δt is the time step size. An explicit method was used for the convective term to avoid solving a nonlinear set of algebraic equations at every time step. The tradeoff was that requirements on Δt must be made to ensure stability.

Using the method of characteristics, the diffusion terms $\mu\nabla^2\vec{u}$ and $\nabla\cdot\vec{u}$ were treated implicitly. A second-order backward differentiation multistep scheme was used with accuracy on the order of Δt^2. This computation was automatically stable, but a set of linear algebraic equations had to be solved at each time step. An iterative solver was used to compute results to these equations.[Fluc]

After simulating the blood flow using Nekton, it was found that velocity gradients were generally low along the direction of the streamlines. The gradients across the streamlines (shear rate) were quite high in areas around the inflow jet and aneurysm vortices. The choice of SEM software was thus based on an invalid assumption. Also, as the geometric model complexity increased, modeling using large hexahedral elements became difficult. Due to the adjacent element conformance requirement, areas of high density (required where strain rates were high) propagated elements unnecessarily into areas of low strain rate.

[1] Flowsolve
[2] Fluid Dynamics International
[3] Computational Fluid Dynamics Services
[4] Fluent Incorporated
[5] Fluent Incorporated
[6] Fluent Incorporated
[7] PDA Engineering
[8] Concentration Heat and Momentum Limited
[9] Fluent Incorporated

In response to our experience with Nekton, it was decided to change the simulation software to Fluent/UNS, a new FVM solver that uses unstructured meshes. This package also offers adaptive meshing, which should decrease the labor required in building a conforming mesh. All simulation results presented in Section 3.5 are from the Nekton package.

3.5 Validation

The early simulated flow data results match qualitatively with those seen on clinical angiography. In the simulated lateral wall aneurysms, we observed distal central inflow zones, proximal lateral outflow zones, relatively slow helical rotation, and a stagnant core region[Har93]. In order to provide quantitative validation of the simulation, flow data was collected in swine aneurysms using endovascular Doppler velocimetry.

It should be kept in mind that there cannot be absolute validation of a simulation. Simulation results can be used in a variety of ways from extracting specific high precision data to gaining insight into general trends. The degree of validation only need match the intended use of the data. Besides the difference in detail required from simulation results, the type of model simulated can affect appropriate validation. Numeric simulations can be used for predicting the outcome of realistic, measurable physical situations or for exploring models that are difficult or impossible to physically implement. In the former situation, validation is fairly straightforward since the empirical validating data is readily available. On the other hand, simulations of models that are difficult or impossible to physically implement are problematic to validate because, by definition, empirical data is sparse or nonexistent. In the worst case it may be necessary to rely on trusting (or independently validating if possible) the underlying models and assumptions and evaluating the ultimate efficacy of the results.

Data from the aneurysm blood flow simulation is to be used to plan and evaluate endovascular treatments. Initially it will be used to gain insight into the general blood flow patterns, to help plan procedures. For this purpose, qualitative validation should be sufficient. The second step will be to simulate specific treatments and the forces they cause on the aneurysm walls so as to predict rupture. In this case, quantitative validation is desirable. Unfortunately, quantitative in vivo endovascular flow data is difficult to acquire. Angiography and Doppler velocimetry provide only regional flow information. In this phase, both qualitative and quantitative (Doppler) validation were pursued. Due to the difficulty in measuring detailed blood flow, the final test of the aneurysm blood flow simulation results will be in the measurable benefits gained in health care.

3.5.1 Experimental Model

The aneurysm blood flow simulation was tested using lateral wall aneurysms surgically constructed in five swine. The in-vivo model consisted of a lateral wall aneurysm located on the common carotid artery of a swine. The aneurysm was surgically constructed from a section of vein. The procedure was performed by members of Endovascular Therapy Service and Leo G. Rigler Research Center at the UCLA Medical Center. Angiographic and endovascular physiologic techniques were used to measure geometric and hemodynamic properties of the aneurysm. The in-vivo swine aneurysm model was chosen based on its geometric similarity to typical human intracranial aneurysms.

3.5.2 Data Acquisition

Digital subtraction angiography (DSA) sequences were performed on the aneurysms at 30 frames per second in lateral, anteroposterior (AP), and two 45 degree angle views. During each study, flow velocity data was gathered using a Cardiometrics Doppler Flowire. The endovascular Doppler guide wire was used to map velocity in the aneurysm and parent artery at various points. The Doppler device allowed velocity to be measured only along the axis of the probe. A microcatheter connected to a pressure transducer was used to measure pressure at several points within the aneurysm and along the parent artery. Gross dimensions of the aneurysm geometry were manually measured. The DSA image sequences were transferred by ethernet into the Picture Archiving Communication System (PACS) system for further processing.

3.5.3 Aneurysm Model

The image sequences were digitally processed to extract the shape profile of the artery and aneurysm in the four views. These profiles and the manual measurements were used to reconstruct a series of slices of the aneurysm and artery geometry, perpendicular to the general axis of the parent artery. A three dimensional geometric model of the aneurysm was then constructed by connecting adjacent slices with hexahedral elements. The geometry of the artery and aneurysm was modeled as inelastic. A piecewise linear model of the pulsatile inflow velocity was constructed based on the Doppler Flowire charts. Newtonian and non-Newtonian blood models were used.

3.5.4 Doppler Trials

Using data from the five swine aneurysms, initial validation was attempted using Doppler strip charts. The position and orientation of the probe was located using biplanar angiographic images. History points were set in the simulation at these positions to record the velocity values over the heartbeat. Then the orientation was used to find the speed component in the appropriate direction. The simulated speed components were then compared to the Doppler strip charts. The comparison was generally unfavorable. It was found that the single point model of the Doppler probe was inadequate, and a more accurate model was necessary. This was especially true since the endovascular Doppler probe is on a soft wire that tends to ride on the vascular wall. The velocity on the wall should be exactly zero, contrary to the strip chart values. The specifications for the Doppler probe used indicates that it samples speeds parallel to the wire in a cone 4.2 mm high with a half angle of 22 degrees. Thus speeds were being picked up away from the wall. In order to simulate this measurement, a virtual tool was created for use in the visualization environment that sampled parallel speeds in the equivalent cone. By positioning and orienting the virtual Doppler probe in the correct location, a simulated strip chart could be created to correspond to the real Doppler strip chart.

Of the five simulated swine aneurysms, one was selected for validation using the virtual Doppler probe. This aneurysm, constructed in swine 895, was measured on April 22, 1994 in the Rigler center of the UCLA medical school. Drs. Cheng Ji and Tarik Massoud performed the procedure and acquired the data. The aneurysm was constructed on the carotid artery using venous material. Manual measurements were made of the artery diameter (.45 cm,) neck width (.3 cm) and length (.8 cm,) as well as the aneurysm height (.85 cm,) diameter and width (both .9 cm.)

Using the endovascular Doppler probe, arterial flow was measured upstream (Figure 3.2) and downstream (Figure 3.3) of the aneurysm. The speed profiles are nearly identical in both positions with a peak of 1.1 m/s and a low of .3 m/s. A piecewise linear model of the curve in Figure 3.3 was used as a basis for the simulation inflow.

Three successful Doppler measurements were made at different positions in the aneurysms. Angiograms in two views were made for each one, to determine the probe's position and orientation. Position I was in the neck, facing towards the top of the aneurysm. The lateral projection is shown in Figure 3.4. The probe was occluded by contrast material in the anteroposterior projection, thus this figure is omitted. Figure 3.5 shows the maximum speed for position I measured as 12 cm/s with systolic retrograde speed of 6 cm/s.

In position II, the probe was looped around the wall of the aneurysm and back into neck, facing out of the aneurysm. Figures 3.6 and 3.7 show the lateral and anteroposterior angiographic views of the probe. Figure 3.8 is the strip chart for position II, which has a peak speed of 8.8 cm/s

Position III was located on the middle of side wall of the aneurysm. Figures 3.9 and 3.10 show the lateral and anteroposterior angiograms of the probe. Figure 3.11 shows the strip chart for position III, and the peak velocity of 7 cm/s and systolic retrograde speed of 5 cm/s

3.5.5 Virtual Doppler Trials

To validate the Doppler data, a virtual Doppler probe was integrated into the virtual visualization environment. This probe implemented a model of the measurements made by the endovascular Doppler probe as per the specification given by the manufacturer. It allowed similar measurements to be made on the simulated data that could be directly compared to empirical results. The virtual Doppler probe appears in the environment as a cone that corresponds to the sampling volume. The surface of the virtual probe cone is color mapped to the axial speed on the surface. Using direct manipulation of position and orientation handles on the tool, it

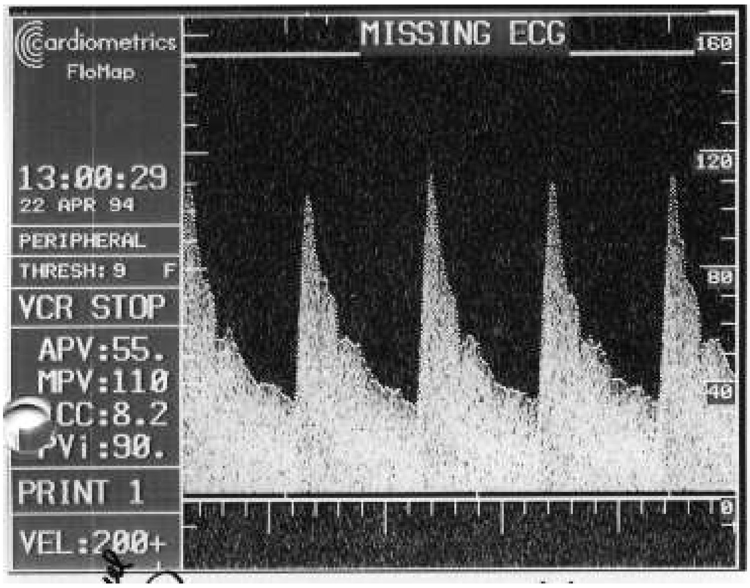

Figure 3.2: Upstream endovascular strip chart for swine 895.

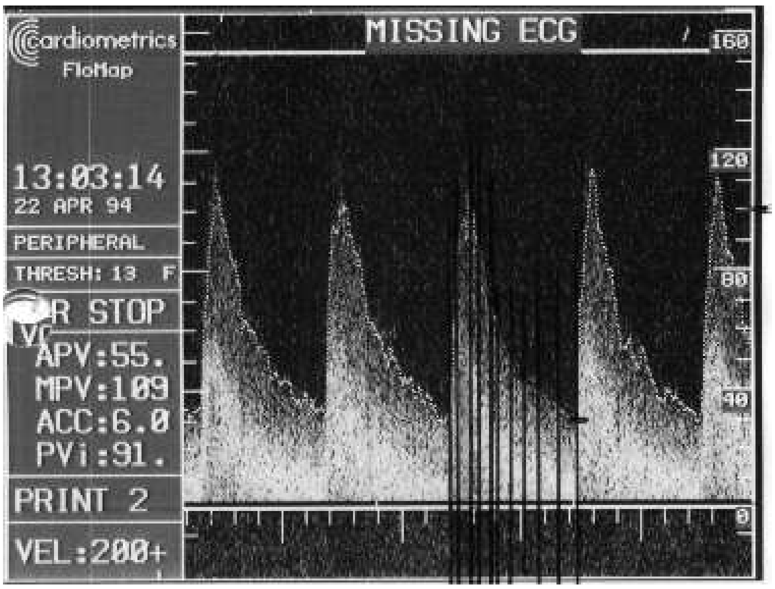

Figure 3.3: Downstream endovascular strip chart for swine 895.

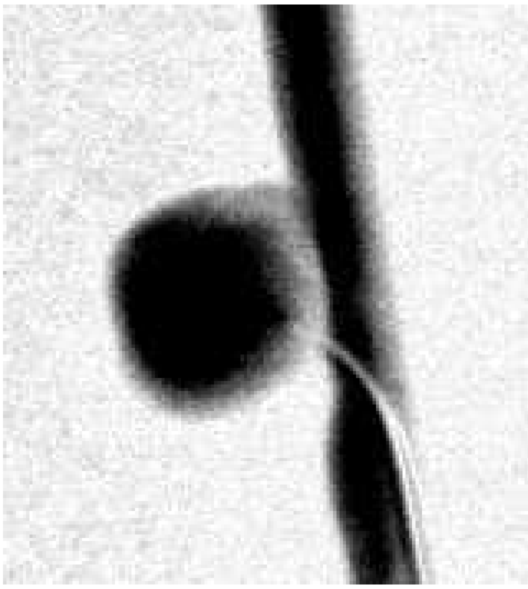

Figure 3.4: Lateral angiogram of swine 895 with probe in position I.

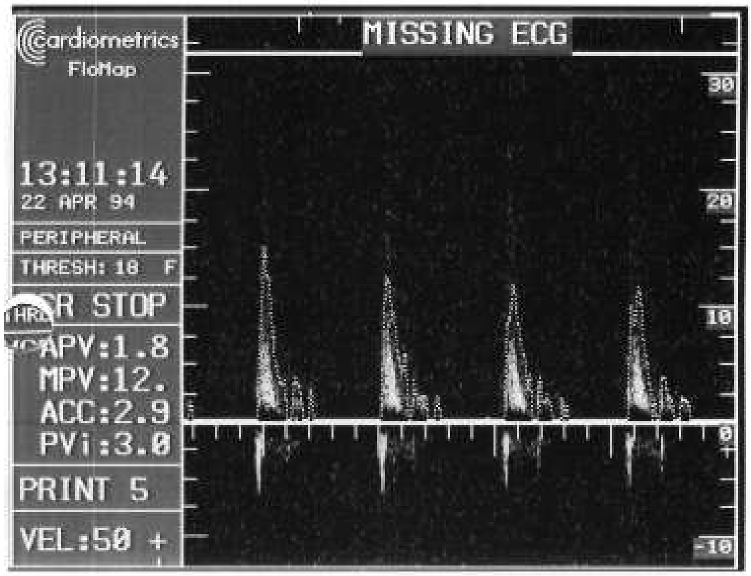

Figure 3.5: Doppler strip chart of swine 895 with probe in position I.

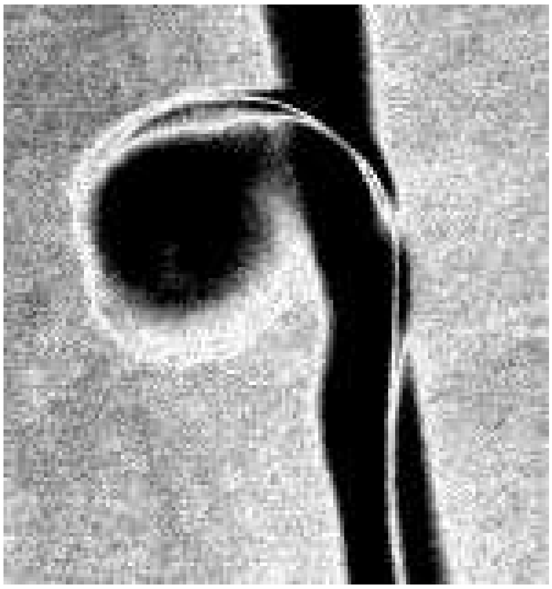

Figure 3.6: Lateral angiogram of swine 895 with probe in position II.

Figure 3.7: Anteroposterior angiogram of swine 895 with probe in position II.

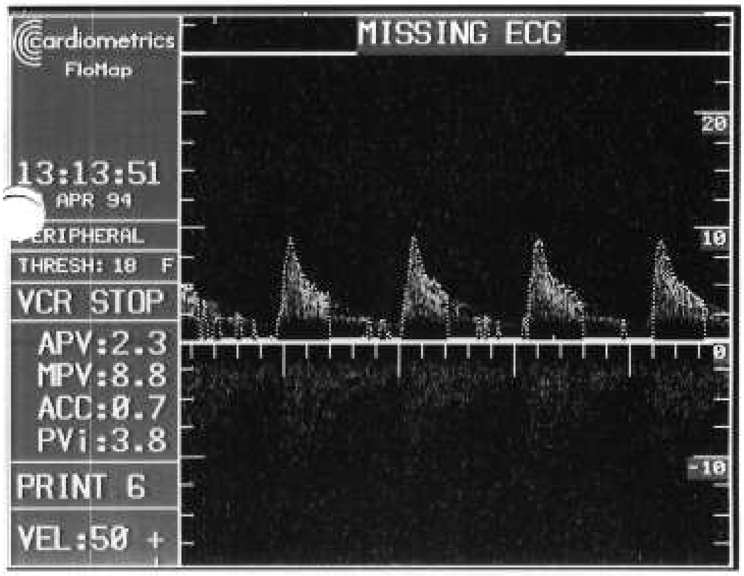

Figure 3.8: Doppler strip chart of swine 895 with probe in position II.

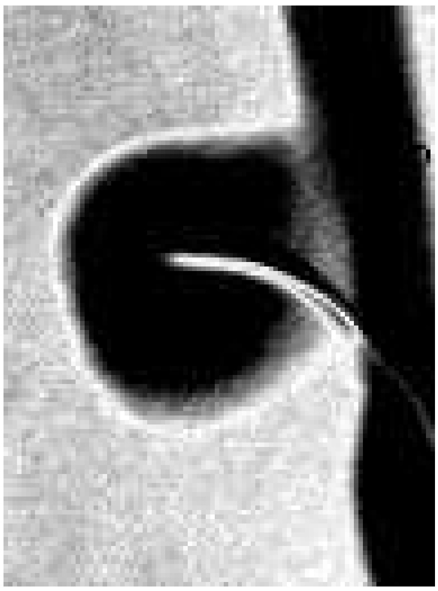

Figure 3.9: Lateral angiogram of swine 895 with probe in position III.

Figure 3.10: Anteroposterior angiogram of swine 895 with probe in position III.

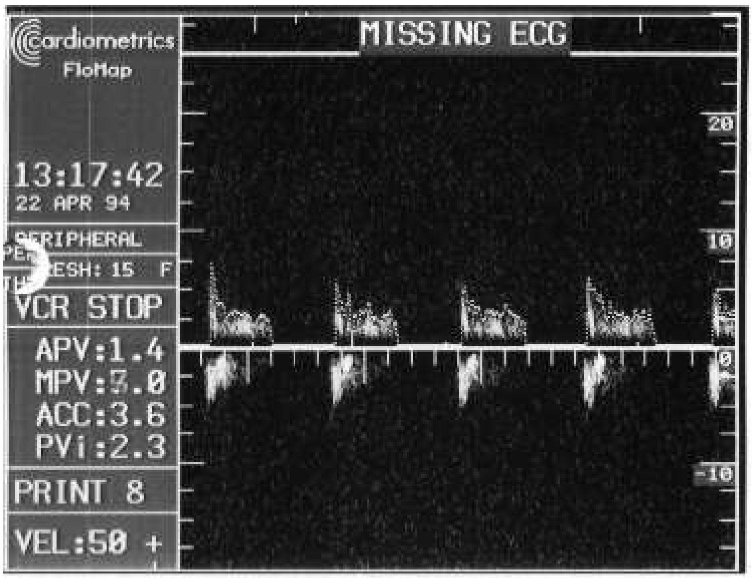

Figure 3.11: Doppler strip chart of swine 895 with probe in position III.

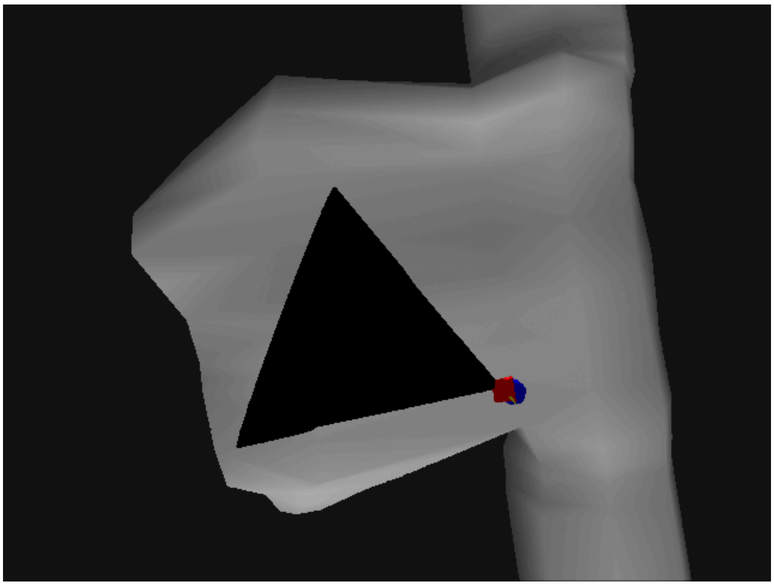

Figure 3.12: Virtual Doppler probe in position I.

can be moved and rotated to a position corresponding to the real probe (as seen on angiography.) Samples of axial speeds inside the cone are made (by querying the Flowfield database) to construct a range of values at each time. These ranges are input to a real time strip chart program that displays a moving graph of the speed range, much like that of the endovascular Doppler probe.

Two blood viscosity model simulation data sets (from swine 895) were tested using the virtual Doppler probe: one Newtonian (5 cp) and one non-Newtonian (5-200 cp.) In each case, the virtual Doppler probe was positioned corresponding to positions I, II, and III as defined for the real Doppler probe (see Section 3.5.4.) Figures 3.12, 3.13, and 3.14 are screen shots of the virtual Doppler probe in the three positions. These three figures correspond to the Newtonian data set, the non-Newtonian set used approximately the same positions which are not shown here.

For each of the six trials (2 models, 3 positions) axial speed ranges versus time were collected so they could be compared to the real Doppler data. Figure 3.15 shows results from both models in position I. The non-Newtonian flow is slower than the Newtonian, roughly half at diastole and three-quarters at systole.

Figure 3.16 compares the models at position II. Again the non-Newtonian flow is slower: approximately one half at diastole and four-fifths at systole. The Newtonian model shows minor retrograde flow.

Figure 3.17 is from position III: the probe inside the aneurysm facing the neck. There are large differences between the viscosity models in this position. The non-Newtonian data shows most blood flowing away from the probe, i.e. out of the aneurysm. The Newtonian data shows some flow moving out of the aneurysm and more coming in. Some apparent instability is seen, especially in the Newtonian case. Since the flow was modeled as laminar, this is most likely numerical instability. This indicates a possible problem in the simulation setup, such as mesh density.

Table 3.1 compares the real and virtual Doppler strip charts as speed ranges at systolic and diastolic phases. The systolic was measured as the maximum value across the heartbeat. The diastolic was measured as the local minimum directly before the systolic point. Generally speaking, the lower speeds of the non-Newtonian model match closer to the real Doppler data in positions I and II. The non-Newtonian systolic data is 1.5-3 times too large while the Newtonian is 2-5 times too large. In position III, however, the Newtonian model comes much closer to matching the bidirectional flow measured in the swine.

3.6 Analysis

Using angiographically located Doppler velocimetry as a validation tool, aneurysmal blood flow was validated with error rates of approximately 300% for a non-Newtonian model and 500% for a Newtonian model. A

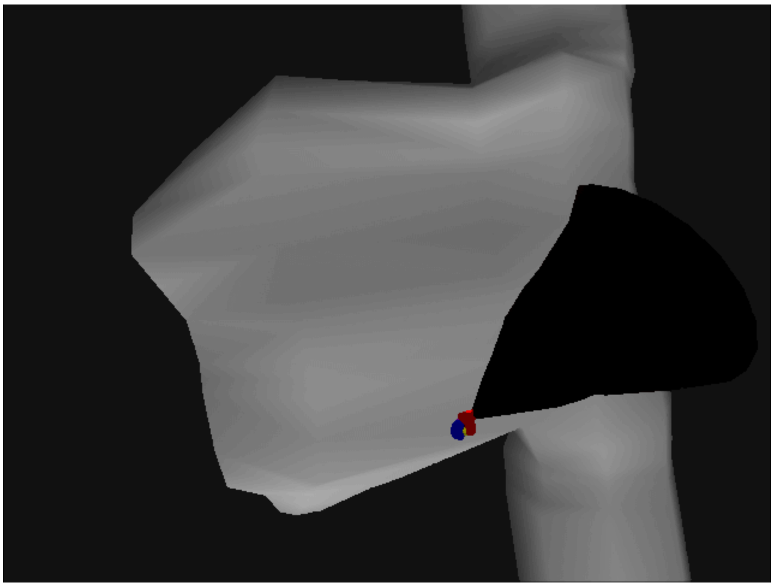

Figure 3.13: Virtual Doppler probe in position II.

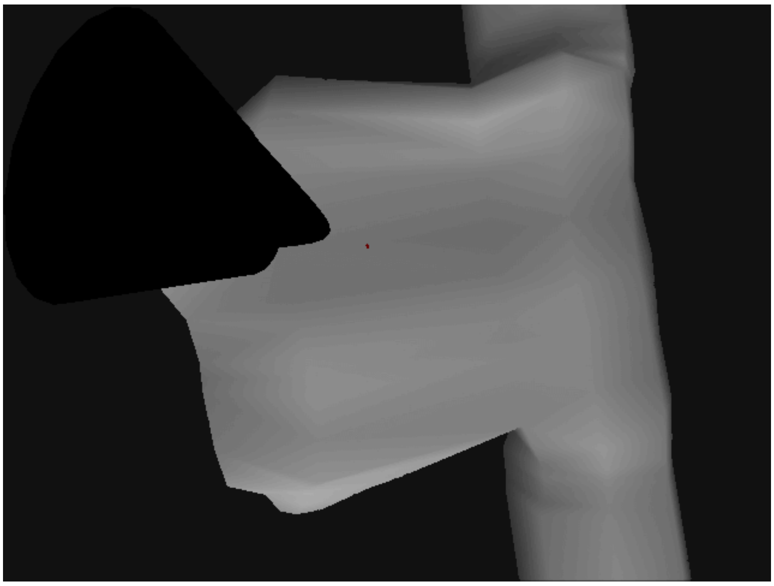

Figure 3.14: Virtual Doppler probe in position III.

	Position:	I			II			III		
Systolic:	Doppler	-6	–	12	0	–	9	-5	–	7
	Newtonian	0	–	57	-1	–	20	-9	–	5
	non-Newtonian	0	–	41	0	–	16	-.3	–	12
Diastolic:	Doppler	0	–	2	0	–	2	0	–	0
	Newtonian	0	–	19	-.5	–	5	-2	–	1.5
	non-Newtonian	0	–	10	0	–	3	0	–	4

Table 3.1: Summary of Doppler validation results. Speed ranges for the real Doppler probe compared to the virtual Doppler using Newtonian and non-Newtonian models. Results (in cm/s) are for three positions examined at systolic and diastolic phases.

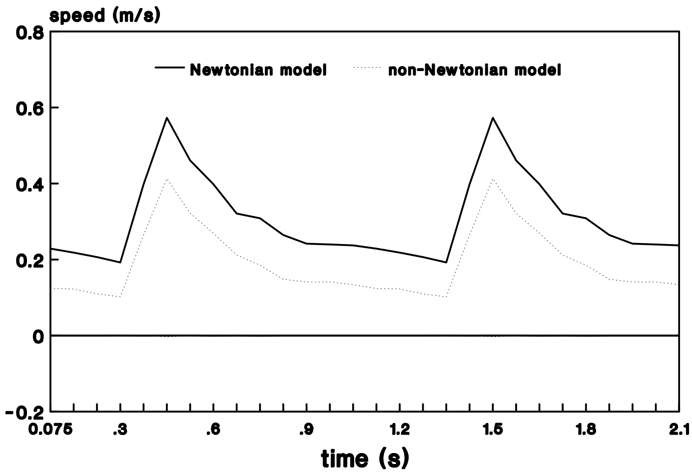

Figure 3.15: Virtual Doppler strip chart of swine 895 with probe in position I, for Newtonian and non-Newtonian models.

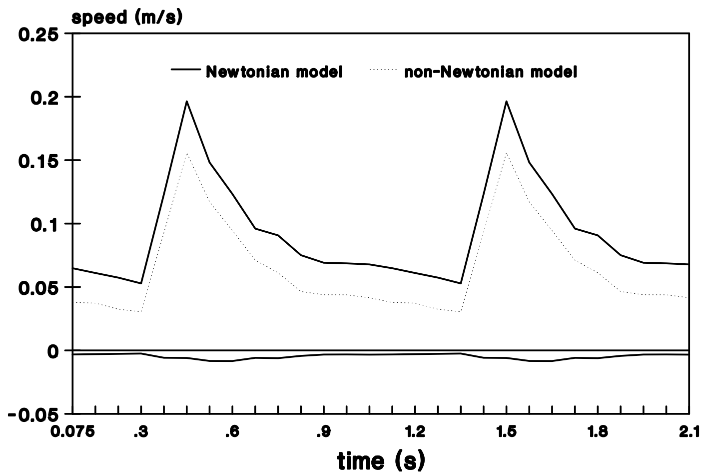

Figure 3.16: Virtual Doppler strip chart of swine 895 with probe in position II, for Newtonian and non-Newtonian models.

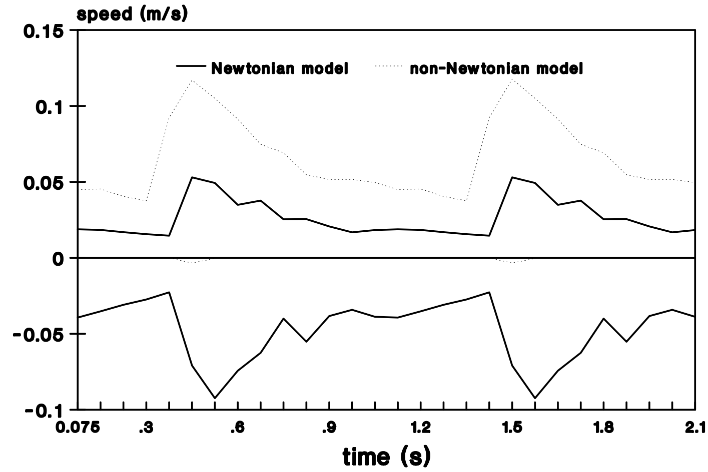

Figure 3.17: Virtual Doppler strip chart of swine 895 with probe in position III, for Newtonian and non-Newtonian models.

maximum of 20% error in arterial blood flow was computed across all viscosity models. The major error therefore occurs in the aneurysm flow. There are several areas of the modeling, simulation, and validation that should be investigated to attempt to reduce the aneurysmal flow errors so as to attain quantitative validation sufficient for clinical use. Furthermore, the accuracy of the validation method itself needs to be examined.

The geometric configuration of the aneurysm neck contributes to the amount of flow that goes into the aneurysm. Moving the distal lip away from the artery will tend to deflect flow, while moving the lip into the artery will tend to scoop more flow into the aneurysm. Using angiography to determine aneurysm geometry makes it very difficult to determine the exact configuration of the neck. In these trials, this determination was mostly guesswork. The quality of the simulation data would most likely improve if more accurate geometric models could be built. Other imaging modalities such as magnetic resonance or computed tomography may provide the details required to construct a high quality geometric model.

While it is apparent that the viscosity model does affect the flow results, it doesn't appear to be the major contributor to the error. Both Newtonian and non-Newtonian models generated similarly inflated aneurysm speeds. Until the general error can be reduced, the effects of the viscosity model are most likely superfluous. Since it appeared to be more accurate inside the aneurysm, the non-Newtonian model should be used for further studies. Once error rates due to other factors can be reduced, the relative effects of the different viscosity models can be more easily studied.

Some numeric instability was detected in the results, indicating that the mesh density might be too low. Since hexahedral mesh generation and refinement were so time consuming, and because it was not easy to evaluate a mesh by hand, constructing satisfactory spectral element meshes was extremely difficult. An alternative to this approach is to use an adaptive meshing scheme. In this way, an initial (usually tetrahedral) mesh is constructed and submitted to the solver. The solver iteratively adapts the mesh density as the solution proceeds. Using such a system, a satisfactory mesh can be automatically built conforming to a particular problem without involving a human operator.

A final area that should be investigated is better empirical validation tools. The endovascular Doppler probe was difficult to locate using angiography. There was substantial guesswork involved in determining the correct probe position and orientation for the virtual probe study. If Doppler studies continue to be used in validation, better methods for locating them (possibly MR or CT based) need to be developed.

While the differences between computed and measured aneurysmal flows seem daunting at first, the computed results did agree with the results of other approaches. The simulation results were found to match in vivo results (locally acquired cineangiography of swine aneurysms as well as [GSPR92]), in vitro results

[LSPR87, NKS84], and other numerical simulations (especially the sophisticated models of [LPR93]). These agreements, together with the general acceptance of the underlying models and methods (e.g. the Navier-Stokes equations and the finite element method) indicate that the large errors are more likely contributed by the validation technique.

Chapter 4

Virtual Reality Visualization

Virtual reality (VR) is a concept with no clear definition and a multitude of competing jargon terms. Artificial reality, cyberspace, immersive interface, synthetic experience, virtual environments and virtual worlds are all terms that are used to describe computer interfaces that are based on some experiential metaphor.

Arguably the purist type of VR interface is immersive. This type of system can be implemented with a head mounted display (HMD) and trackers that measure body and head position, and hand gestures. The effect is a first person experience of existing in and interacting with a computer generated world. At a lower level of immersion is so called through-the-window or fish tank VR. In these systems, a user can imagine that a display screen is a window into a three-dimensional space. Such systems use stereoscopic displays and head tracking to allow direct viewing of three-dimensional structures. A third type of system uses a second person perspective of the user in the virtual world. A video camera captures the outline of a user and the computer integrates this as an interacting entity in the graphical world.

Virtual reality is a rapidly growing and dynamic field. It is difficult to provide a clear definition of what is and is not virtual reality. Instead, interface systems can be considered as scattered along a spectrum from not-VR to very-VR. Howard Rheingold suggests two traits of an interface that are important to VR: immersion and navigation[Rhe91]. Immersion is the sense of presence a user has in a virtual world. There are periods of suspended disbelief where a user is absorbed in the task and forgets the interface. Navigation is the ability to move about and freely explore a virtual world. Burdea and Coiffet[BC94] add two other traits that contribute to a VR interface: interaction and imagination. By imagination they refer to the extent by which the underlying simulation and interface solve a particular problem, which is dependent on the imaginations of virtual environment designers. With this graded approach there is no cut off point for what VR is and no dependence on a specific technology or device.

4.1 Virtual Environment Design

The user interface was designed by considering the tasks and goals of the user, as well as the nature of the data. The decision to use an immersive virtual environment for blood flow visualization was made early on in the design process. This decision was made because:

1. the three dimensional nature of the data could be mapped directly to the interface;
2. an intuitive direct manipulation control system could be implemented; and
3. the immersive aspect would help focus attention on the task at hand.

Using a virtual environment, we hope to exploit natural human skills such as visual (three-dimensional spatial processing capabilities,) motor (direct manipulation of objects,) and verbal (voice commands) to best communicate information from the computer to the user and vice versa.

With this foundational concept in place, the user characteristics, usability requirements, and a task analysis were analyzed to provide guidance in further design. In general, interface design followed a user-centered philosophy, as per [RPM95]. This philosophy seeks to actively match the interface to the particular users and tasks that it serves. The process covers the following steps:

1. Identify end-users and their usability requirements.
2. Determine users' tasks and mental models.
3. Involve end-users in analysis, design, and prototyping.
4. Conclude with end-user evaluation of prototype usability.

4.1.1 User Specification

Before designing a usable VR blood flow visualization system, the users of the system had to be identified and their characteristics understood. By keeping in mind the attributes of the end-user, the designer can better create an interface that serves the real users. This also helps avoid designs that may be personally pleasing to the designer yet inappropriate to the users.

One issue that must be considered when designing a VR system for medical use are the skills and goals of the physician. The goal of the physician in coming to the system is to get help in perception. The ultimate goal is to make an educated decision, or successfully accomplish a procedure. The user can be expected to have high skill and knowledge in the task domain, but not necessarily to be experienced with computers. Due to the importance and complexity of the perception task, it is reasonable to expect physicians to participate in significant training. To have confidence in the images, the physician should be aware of the nature of the processing.

To be useful, the system must be accepted by the physicians who are to use it. In a study of diffusion of computer applications in medical settings[AJ87], it was found that the particular nature of physicians' work and attitudes are a major reason for non-acceptance of computers. Doctors as a group are independently minded and tend to be resistant to external forces of change.

Another study [Kap87] considered the effect of physicians' values on system acceptance. It was concluded that for a system to be accepted, quality of care must be increased. Outcomes must be improved, risks reduced or procedures made more pleasant. Otherwise systems are likely to be rejected. The study found that while physicians appreciate more scientific information, they do not appreciate systems that attempt to replace the artistic aspects of medicine. Doctors valued the role as chief decision maker, the autonomy to deviate from recommended courses, and the physician-patient relationship.

Major system problems identified in [LC90] were informatics coherence and psychological acceptance. To be successful, a system must be considered as a helpful tool by both physician and patient.

Kubler and Gerig [KG90] point out the danger of over-processing image data. Even though all images are artificial, there is a line drawn between so-called raw or original images and overly filtered images. Designers must be careful that the data visualization is not too abstract.

Anderson [AJ87] identifies the function of medical written records as another consideration that should be taken into account when designing systems. The placement of text, symbols, and margins on a document have meanings that supplement the text. The layout of a medical document is used as a factor in decision making. A system that does not accommodate this implicit information will frustrate doctors by impeding the normal decision making process.

The function and scope of a computer application will have a psychological effect on physicians that can determine its success or failure. To be accepted, a VR medical imaging system should improve health care and respect the expertise of the doctor.

Three user classes have been envisioned for the system. These three types of user will be involved in:

1. Visualizing flow and stress fields, and anatomy of an aneurysm,

2. Understanding the effects of a proposed intervention, and

3. Learning to perform intervention procedures.

These three types of users will have different usability requirements. In this study, analysis will focus on the first user class: the radiologist who is visualizing simulated blood flow from a clinically encountered brain aneurysm. Figure 4.1 highlights the radiologist's characteristics and corresponding design implications.

Radiologist	Notes	Design implications
Type of use	Direct	
Computer Experience	Scanners, Macs	Menu and GUI systems understood
Frequency of use	0-2 hours/week	Shortcuts and flexibility not crucial
Mandatory/discretionary	Discretionary	Quick learning, high confidence
Other systems in use	Digital scanners	Standard views, controlled indirectly by voice
Education/intellect	High	
Motivation/goals	High/Decrease morbidity	

Table 4.1: Radiologist characteristics and implications for interface design.

4.1.2 Usability Specification

There are several different settings where VR medical visualization might occur. Two examples are during surgery or diagnosis and surgical planning. Each setting has its own special requirements.

In a surgical environment, sterilization is a factor. There may be limited room in the surgical theater for equipment. User stress levels will be high, and system response time must be fast. The interface should be simple, and it should not be possible for user errors to cause injury to the patient. The display system should allow rapid switching between image perception and surgery. An immersive head mounted display would probably be inappropriate because of this constraint.

A system for diagnosis and planning should have navigation and data exploration tools to aid in the perception task. The physician should be under low stress and system response time is not life critical. It should be easy to bring up any information relating to the case, and to add data to the case file directly through the interface. For cooperative work, multiple physicians could explore the dataset. Again, this would raise another range of issues. This application would most benefit from full immersion. The three-dimensional data exploration task would exploit the stereoptic immersion and natural manipulation qualities of a head mounted display and three-dimensional input device combination. Interaction with the case history would have to occur strictly through the head mounted display and three-dimensional input device, a keyboard would be practically unusable. Voice commands or virtual world menus may suffice. Any system would have to be tested to determine what effect the interface has on the decision making process.

The issues mentioned are only a few specific to the domain and interface type that have to be considered. Some other more general questions are: weight of equipment, hand and eye strain, anthropometry, required motor skill and visual acuity, method of navigation, available exploration tools, gestures, menu layouts, icon appearance, and color selection.

The usability requirements of the system provide important goals to keep in mind while designing the interface. In addition, they act as a benchmark for judging the system based on the usability evaluation. The critical usability requirements of the VR blood flow visualization system are:

1. Movement around and within the aneurysm should be natural and easy.
2. Virtual tools should be intuitive and easy to control.
3. The physician should be able to gain insight into the flow field structure.

More generally, the following interface design principles were followed (derived from Nielsen's ten principles[RPM95] and Shneiderman's eight[Shn92].):

4. Interface consistency should be maintained.
5. Short cuts should be provided for experienced users.
6. Feedback for action completion should be given.
7. Closure should be shown for task sets.
8. Errors should be prevented or easily recovered from.
9. Actions should be reversible.
10. User should feel in control of system.

11. User memory load should reduced.
12. Clutter should be minimized.

4.1.3 Task Analysis

The task model is limited to those tasks involved in investigating and exploring the flow field. A hierarchical task analysis (HTA) technique is used to divide the goal into finer subtasks.

- Understand aneurysm flow field
  - Select aneurysm
    - Choose patient
    - Choose simulation run
  - Get overview of aneurysm geometry
  - Choose region of specific interest
  - Move to region of interest
    - Move to general region
    - Fine tune view
  - Select appropriate tool
    - Select tool
    - Activate tool
  - Manipulate tool
    - Move tool into position
    - Adjust tool parameters

It can be seen from this task analysis that the major interface design hurdles are how to implement:
1. navigation (coarse and fine), and
2. tool control (selection and manipulation).

In order to make the design decisions sufficient for building a prototype interface, the user and usability specifications were considered in the context of the task analysis. Some lessons were also learned from experience with an early ad hoc prototype.

4.1.4 Interface Design

Starting from a virtual environment based design constraint, the major parts of the interface design: navigation, and tool selection and control were explored. The design was then implemented and tested.

Navigation

To freely explore the flow field, the user must be able to move easily about in the virtual space. Also it should be impossible for the user to get lost in the space. The initial navigation system used the two dimensional mouse to control virtual motion of the user. Combinations of mouse positions and button presses were used to translate, pitch, roll, and yaw. While it was possible to learn the somewhat esoteric mapping after some training, it was decided that this technique was unacceptable. Two major factors to this decision were that the mapping was unintuitive and it required the user to keep a hand on the two-dimensional mouse. Another disadvantage of this technique was that it was easy to get lost, or to lose sight of the three-dimensional cursor (virtual hand.) Using the keyboard for navigation had similar shortcomings.

It was decided to enable hands free navigation by using voice recognition. For general motion, the user can use voice commands such as "move forward," "turn left," and "stop" to initiate and halt translations and rotations. Other voice commands allow the user to smoothly and automatically move to certain positions. These positions can be preset to correspond to standard views, e.g. "goto lateral" (view). Positions can also be set by the user at run time, e.g. "set alpha," followed later by "goto alpha." The voice command "center hand" brings the virtual hand cursor automatically in front of the user. By using this voice command strategy, the hands are free to directly manipulate the virtual objects in the environment. The preset and programmable positions should allow the user to become quickly oriented by automatically returning to a familiar position.

Tool Control

The tool design methodology chosen yields tools with all controls integrated into the tool itself. Menu selection, form fillin, and command languages are typically used to control visualization tools. Some use a manipulable graph of linked processes to build custom visualization tools[LAC+92]. All of these methods divide the tool into two separate parts: the controls and the display. This separation of control and form tends to distract the user from the task at hand. The design technique decided upon is to combine these two roles (visible representation and control) into one directly manipulable object. Control will be performed using integrated three-dimensional handles[CSH+92, ZHR+93] and supported using object-oriented visualization environment design techniques[TGB+90, GBT93, FH94]. The handles will have consistent functions and appearances across different tools. Tool selection will be controlled via voice commands, thus reducing the input devices to a three-dimensional mouse and a microphone.

4.2 Interface Implementation

A prototype of the interface design was constructed so that the design concepts could be evaluated by the end-users. The system consisted of a workstation based VR platform and an object-oriented application for implementing the virtual environment.

4.2.1 Platform

A VR interface was built to allow interactive and immersive visualization of the simulated aneurysm blood flow. A Sun SPARC 10 with a ZX graphics accelerator (Sun Microsystem Inc., Mountain View, CA) with Stereographics CrystalEyes shutter glasses and Logitech 3D Mouse was used as a hardware test bed. The shutter glasses allow a user to view three-dimensional images that appear to float within the workstation monitor. The 3D mouse can be moved and rotated in front of the monitor to allow manipulation of virtual tools. The Sense8 WorldToolKit VR library was used to manage the devices and graphical objects. This library allows graphical objects and their behaviors to be defined at a high level. The technical details of the VR simulation loop (e.g. sensor control, object updating, view maintenance, rendering) are controlled by the library. GNU C++ was used as the development language of both the underlying flow data model and the visualization environment.

4.2.2 Class Hierarchy

Early prototype virtual environment were developed using a structured programming approach. This approach was deemed unsuitable for further development because it failed to exploit the commonality between objects through code reuse. As many tools and new capabilities were added to the interface, manual maintenance of consistency and commonality became untenable. For this reason, an object-oriented approach was adopted that would make commonality explicit through inheritance. The complete source code for the Virtual Object class library is available as an appendix to a UCLA Computer Science Department 1996 technical report.

The lowest level objects in the hierarchy were typically those that were visibly manifested in the virtual environment. This included aneurysm surfaces with various color mappings (classes PressureSkin, PlainSkin, StrainRateSkin), various cutting planes (SpeedCuttingplane, AbsStrainRateCuttingplane, NormSpeedCuttingplane), a virtual Doppler probe (SpeedDoppler), tool handles (PosWidget, OriWidget), and a hand for manipulating handles (Hand).

Most of the functionality of the visible objects is inherited from higher level objects. For example, the SpeedCuttingPlane is solely responsible for determining the value of the speed at a certain time and position. It inherits from Cuttingplane which is responsible for constructing and controlling the tiles of an arbitrarily colored cutting plane. Cuttingplane in turn inherits from StatObjTool which represents any color mapped object with a fixed geometry. StatObjTool inherits from both WTobj and FlowVisTool. WTobj is an abstraction of a rendered three-dimensional object that is implemented using Sense8's WorldToolKit library but could be reimplemented using any 3D graphics library. FlowVisTool represents any tool for visualizing a three dimensional transient flow. It controls access to the underlying flow field and maintains an arbitrary collection of tool handles. FlowVisTool inherits from the highest level object, VirtualObject. VirtualObject abstracts those things in common with all objects, such as whether they are active, and what their position and orientation is.

This object-oriented strategy allows new tools to be added quickly through inheritance of existing classes. For example, a PressureCuttingPlane could be added by writing a class that computed pressure at a point (a triviality) and inheriting from CuttingPlane. Figure 4.1 provides a simplified Booch style class diagram. The clouds represent classes and arrows represent inheritance ("is-a"), pointing from child to parent. Solid dots correspond to the "has-a" relationship, hollow dots to the "uses" relationship. The dots are on the side of the ray that "has" or "uses."

Control of the visualization objects (through handles, or Widgets) is implemented using a generic template class called DynVar (for Dynamic Variable.) A DynVar allows a class parameter to either take on a locally set value, or take on the value of a remote DynVar of similar type. The remote DynVar could in turn dereference another DynVar. Thus a graph must be traversed to determine the value of a particular variable. This graph can be changed dynamically at run time to create different dependencies. The current topology of the graph is transparent to the object, which simply requests the current value of the DynVar. This technique allows the connections between objects to be dynamically changed without altering the basic code of the objects. For example, a cutting plane sitting in space has some position, as well as an attached PosWidget, whose position is linked (through a DynVar) back to the cutting plane position. Thus if the cutting plane where to move (say through a rotation), the PosWidget would follow it and remain attached. If a Hand object grabs the PosWidget, the links are changed. The position of the PosWidget is linked to the Hand, and the position of the cutting plane is linked to the PosWidget. By traversing the links, the cutting plane is effectively position linked to the Hand.

4.2.3 Environment

After precomputing the blood flow within the aneurysm and artery over one heart cycle, the simulation data was visualized using the VR environment. This involved first organizing and indexing the large dataset, (Section 2.3) then constructing virtual tools that allow interactive and immersive investigation of the blood flow patterns.

A set of preliminary virtual tools was implemented to allow testing of the hardware platform, flow model, and interface design. A user can navigate around and within the aneurysm skin. The skin can be color mapped to animated pressure or strain that follows the heart cycle. Additionally, a wire frame model model can be brought up to provide a spatial reference for other tools. A color mapped cutting plane can be manipulated to investigate either maximum speed, normal speed, or surface strain within the aneurysm and artery in any orientation. These cutting plane tools can be used to identify inflow, outflow, and stagnation zones. A "virtual Doppler probe" models an endovascular Doppler probe that measures velocities in a conical volume. The probe measures flow value ranges in the cone. A geometric representation of the cone is displayed, and color mapped flow values are displayed on the surface. A real time strip chart appears similar to the strip chart on the endovascular probe. All of these tools are animated to follow the changing conditions as the heart beat pulses through the artery. The cutting planes and virtual Doppler both have attached control handles for directly manipulating position (red cube) and orientation (blue sphere.) The cutting planes additionally have a handle for controlling color map range (green cylinder,) so that low flow values (as typically occur in the aneurysm) can be explored.

Figure 4.2 shows a cutting plane display color mapped speed. The plane is positioned in the neck of the aneurysm with the normal pointing along the axis of the artery. The fast arterial flow and aneurysm jet are visible. The position handle and orientation handle are visible on the bottom of the tool. Figure 4.3 shows flow strain rate in the the same general position. High stress can be seen near the walls of the artery and the center of the neck. The same tool positioned in the artery outflow is seen in Figure 4.4. As is expected in general pipe flow, high stress occurs close to the walls. Figure 4.5 shows the artery and aneurysm geometry,

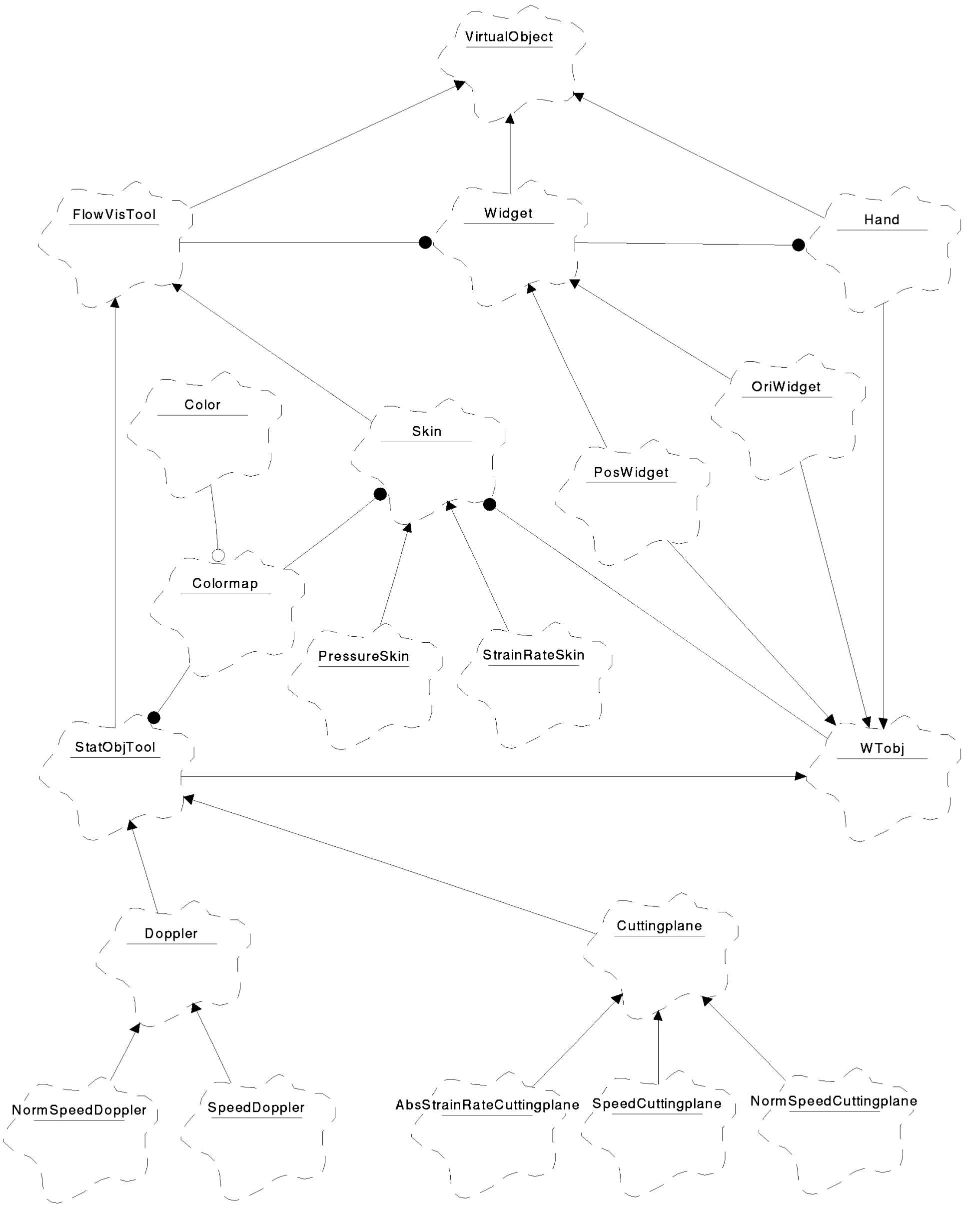

Figure 4.1: Virtual object class library diagram.

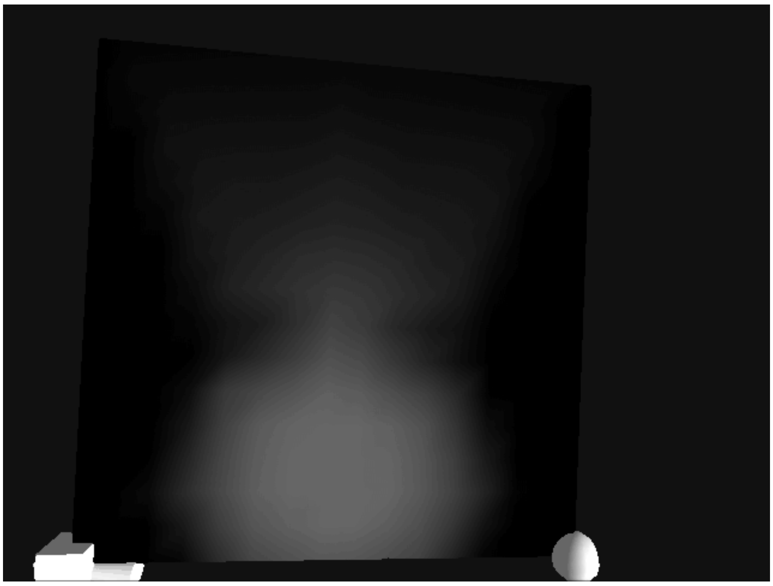

Figure 4.2: Maximum speed cutting plane in aneurysm neck.

color mapped with surface stress. In the virtual environment, this color map goes from black to red to yellow, using a heat style color map to represent strain rate. In mapping this to a gray scale image for purpose of publication, the intensity of yellow (highest) values was exaggerated to emphasize the high values found on the distal neck.

As described in Section 4.1.4 voice commands were implemented to control views, navigation, and tool selection.

4.3 Usability Evaluation

In order to evaluate the suitability of the interface designed, selected users were given a brief introduction and training on the system, then asked to perform two scenarios. A reference sheet with voice commands was made available (see Figure 4.6). The users were encouraged to make comments and ask questions while using the system. Difficulties were recorded as the evaluation proceeded. After completion of the scenarios, each user filled out a short survey. The scenarios were:

1. User locates maximal aneurysm surface strain, and moves close to study that region.
2. Using the same aneurysm, user identifies the aneurysm inflow and outflow zones within the neck.

The first scenario was used to exercise the navigational components while the second emphasized tool control.

4.3.1 Survey

A user survey was used to rate general satisfaction with the system, to measure adherence to the usability specifications (see Sect. 4.1.2) and to collect comments and suggestions. After completing the two scenarios, the user survey was given (Figure 4.7.) Once the questionnaire was completed, users were asked to answer and discuss the following two questions:

1. How did you like using voice commands? Would you rather have used another input device such as a mouse, keyboard, or joystick to navigate?
2. How did you like directly manipulating tool parameters with attached controls and a 3D mouse? Would you rather have used another input device such as a mouse, keyboard, or joystick to control the tools?

Responses to these questions as well as comments made while performing the scenarios were documented.

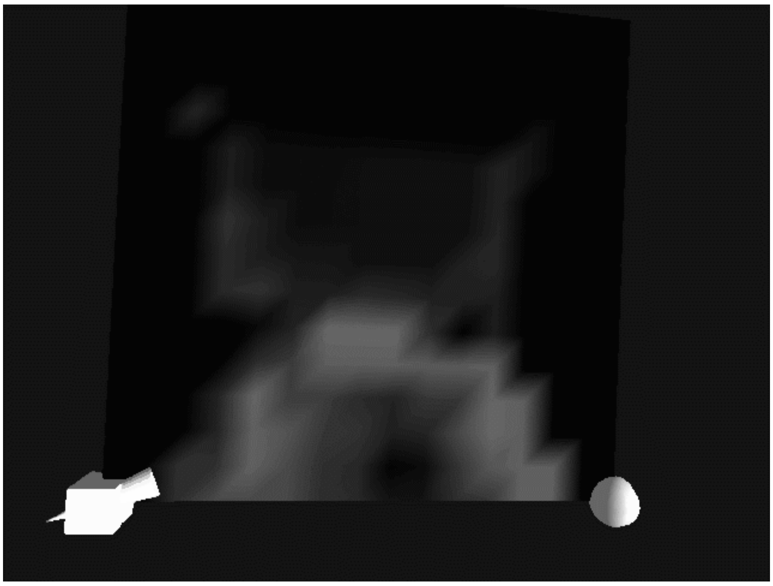

Figure 4.3: Surface strain rate cutting plane in aneurysm neck.

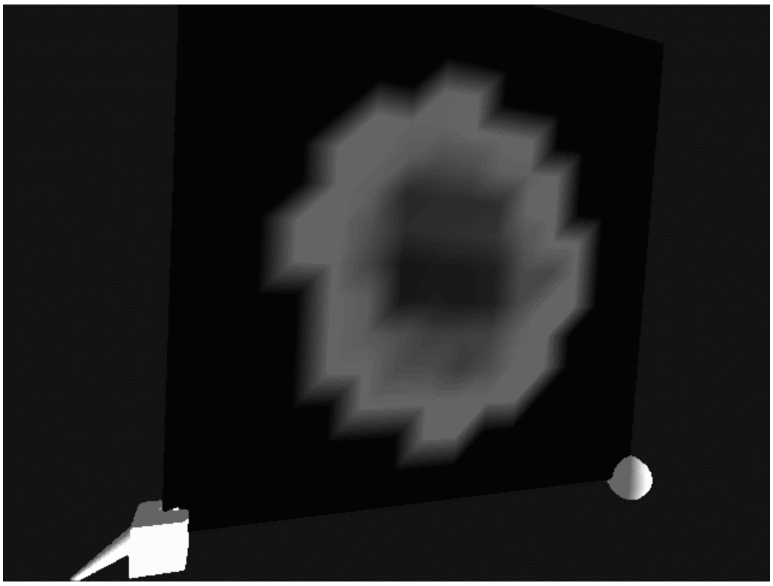

Figure 4.4: Surface strain rate cutting plane in artery outflow.

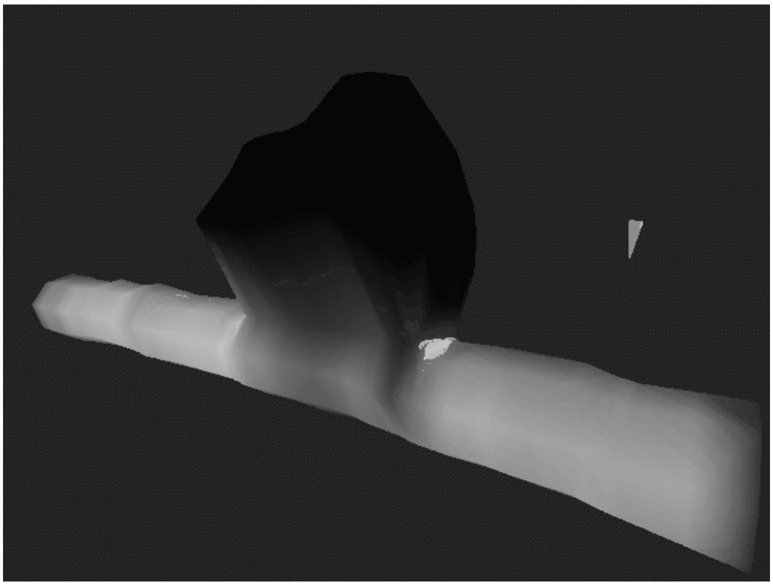

Figure 4.5: Aneurysm and artery surface strain rate.

```
Virtual Aneurysm Voice Command Reference

Translation:

              MoveUp
            MoveForward
  MoveLeft     Stop       MoveRight
             MoveBack
             MoveDown

Rotation:

  RollLeft    TurnUp    RollRight
  TurnLeft     Stop     TurnRight
             TurnDown

Views:

  GotoAP    GotoLateral
  SetAlpha  GotoAlpha

Tools (command toggles):

  PressureSkin       StressSkin               WireFrame
  SpeedCuttingPlane  NormalSpeedCuttingPlane

Misc.:

  CenterHand
```

Figure 4.6: Virtual aneurysm voice command reference.

```
Virtual Aneurysm Questionnaire:

Please respond to each of the following ten statements by circling
a number between 1 (strongly disagree) to 7 (strongly agree).

1.  The system was easy to use.
      1 2 3 4 5 6 7

2.  The system was powerful enough to be useful.
      1 2 3 4 5 6 7

3.  The interface was consistent.
      1 2 3 4 5 6 7

4.  Adequate feedback was provided to verify my actions.
      1 2 3 4 5 6 7

5.  Errors were difficult to make and easy to correct.
      1 2 3 4 5 6 7

6.  I felt like I was in control of the system.
      1 2 3 4 5 6 7

7.  There were not too many things to remember.
      1 2 3 4 5 6 7

8.  Movement around and within the aneurysm was natural and easy.
      1 2 3 4 5 6 7

9.  Control of the virtual tools was intuitive.
      1 2 3 4 5 6 7

10. I was able to gain insight into the flow field structure.
      1 2 3 4 5 6 7


Please make comments below about specific problems you encountered
while using the system, and suggestions for improvement.
```

Figure 4.7: Usability evaluation questionnaire.

Measure	Scores				Average
1. Ease of use	5	2	3	5	3.7
2. Powerful enough to be useful	6	5	7	6	6.0
3. Consistent interface	6	2	3	6	4.2
4. Adequate feedback	4	3	4	4	3.7
5. Easy to correct errors	4	2	3	3	3.0
6. User in control	5	3	4	5	4.2
7. Low memory load	6	6	6	7	6.2
8. Natural and easy movement	5	4	2	3	3.5
9. Intuitive tool control	4	4	3	7	4.5
10. Flow field insight	5	5	7	6	5.7

Table 4.2: Summary of usability questionnaire results. Results range from 1 (worst) to 7 (best).

Difficulty	# of users
Problems with certain voice commands	4
Difficult to grab handles	3
Can't stop in middle of goto	3
Frame rate too slow	2
No numerical coordinate feedback for tools	2
Can't resize cutting plane	2
Can't speedup gotos	1
Arm getting tired using 3D mouse	1
Tool doesn't start in front of you	1
Some lag on voice command recognition	1
Difficult to control virtual mouse	1
Difficult to gauge relative depths	1
Can't change color map on skin	1
Would rather have user fixed, and move aneurysm	1
Can't goto axial view	1
Rotations also translate	1

Table 4.3: Summary of difficulties encountered during usability evaluation.

4.3.2 Results

Four radiologists were selected to evaluate the system. None of the users had used the system before, or had experience with VR interfaces. Each of the four physicians was able to complete the training, scenarios, and surveys in less than one hour.

The questionnaire (Figure 4.7) was designed to measure the interface adherence to both the general and specific usability requirements listed in Section 4.1.2. Table 4.2 summarizes the individual and average scores for each of the ten questions.

Table 4.3 summarizes the various difficulties reported while evaluating the system, and the number of users who encountered them.

In response to the question about navigation using voice commands, only one user preferred voice commands over other input techniques. Two users reported that they would prefer to use a joystick for navigation. The last user liked voice commands for bringing up preset views, but would rather not navigate preferring to manipulate the aneurysm itself.

One of the four users preferred the three-dimensional mouse for directly manipulating the virtual tools. One user stated a preference for a spaceball (a type of joystick with three translational and three rotational degrees of freedom). One user would prefer controlling tools using a joystick or spaceball and an auxiliary window menu system. The final user preferred a joystick or spaceball for general manipulation in addition to constrained movement or rotation along specified coordinate axes controlled by keyboard.

4.4 Analysis

Based on the questionnaire results (Table 4.2) it can be seen that the best parts of the interface were that the user memory load was low, that the system was powerful, and that insight could be gained into the flow field. This suggests a potential for eventual acceptance in the medical community, as it was felt by the users that some variant of the system would be useful in a clinical setting. The basic concept of VR aneurysm blood flow visualization was appealing to the users, though the prototype was found to have several difficulties.

The biggest weaknesses in the system (based on the questionnaire) were that errors were difficult to correct, navigation was unnatural, the system was generally difficult to use, and inadequate feedback was provided. The major source of these problems can be traced back (from comments and discussion) to difficulties in using the 3D mouse for direct manipulation of tool controls, and using voice commands for navigation. Problems with the 3D mouse included fatigue at holding it up, difficulty with precise positioning (due in part to low frame rates, which dropped below 5 Hz), and problems resolving relative depths of the 3D cursor and tool handles. While using voice commands for bringing up preset views appeared promising, most users did not like general navigation by voice which was considered too slow and unnatural. The majority of users indicated a preference for joystick, spaceball, or keyboard commands to replace both the 3D mouse and navigational voice commands.

The results of the usability study suggest that a fully immersive blood flow visualization system may not be the most appropriate for clinical use. Some VR aspects of the interface were popular such as the three-dimensional graphics, interactivity, and direct manipulation. The immersive components of embedding all of the controls into the interface, and using a first person perspective (moving the user rather than the data) were not as popular. A next generation system might explore a hybrid of a VR interface with a normal window-icon-menu-pointer (WIMP) interface. While such a hybridization would probably decrease the immersiveness of the interface (and thus the task focus of the user) it could exploit the strengths of both interface styles. From the VR style would come 3D (stereoptic) data displays, interactivity, and possibly direct manipulation and navigation. The WIMP style would contribute easier management of complex functions (e.g. through menus), additional flexibility in communicating the state of the application (e.g. displaying coordinates), and allow more direct mapping of lower dimensional parameters (such as color map range or tool size) to simpler input devices. The conflict between these interface styles might be lessened by integrating the windowing portion of the display smoothly into the VR section, for example by using a “dashboard” metaphor.

Chapter 5

Conclusions

5.1 Overview

The purpose of this work was to explore the application of computer technologies such as numeric flow simulation, scientific visualization, and virtual reality to problems in medical care. A system was designed, implemented, and evaluated for immersively visualizing simulated blood flow in brain aneurysms. The specific problem we attacked is how to provide insight into aneurysm blood flow conditions to radiologists that will help in the clinical treatment of brain aneurysms.

Several models were used as a basis for the simulation and visualization. Three dimensional modeling of rigid lateral wall aneurysm geometry was performed based on digital subtraction angiography. Pulsatile blood flow was modeled using the Navier-Stokes equation for laminar fluids. Both Newtonian and non-Newtonian blood viscosity models were used. An object-oriented flow data model was implemented for fast access to simulation data.

Aneurysm blood flow was simulated using transient three dimensional computational fluid dynamics software on workstations. Swine aneurysm models were constructed and used for validation studies. General blood flow patterns (distal central inflow zones, proximal lateral outflow zones, relatively slow helical rotation, and a stagnant core region) were validated using cineangiography. Preliminary quantitative validation was performed using endovascular Doppler probe data.

A virtual reality flow visualization interface was designed using a user centered philosophy. Interface distractions were minimized by using directly manipulable flow visualization tools and voice commands for navigation. The interface was implemented using an extendable object-oriented class library. Physician surveys were used to evaluate the design concepts.

5.2 Discussion

Reflecting back on the aims of the dissertation, (see Section 1.4), this work hoped to address the following three questions:

1. How can clinically encountered brain aneurysm geometry, blood flow, and simulated flow data be successfully modeled for subsequent use in a simulation/VR visualization system?

The approach used was to reconstruct geometry manually from digital subtraction angiography images, blood flow using non-Newtonian, pulsatile Navier-Stokes and continuity equations, and simulated flow data using an optimized, indexed, object-oriented data base.

It was found that manual reconstruction from biplanar DSA was a generally inadequate technique for geometry modeling, due to the limited projectional information, the difficulty in resolving neck configuration, and the intensive manual effort required to generate meshes. Work has started to reconstruct brain aneurysm geometry from three-dimensional modalities such as magnetic resonance angiography (MRA). Using a marching cubes based algorithm, it should be possible to automatically construct a polygonal surface mesh for the aneurysm from MRA images. Such a mesh can be used as input to an automatic volume mesh generator. The tetrahedral volume meshes generated should be suitable discretizations for numeric simulation. Using such a technique, it should be possible to completely automate the geometry modeling process, making clinical use of the system more viable.

The laminar, transient three-dimensional Navier-Stokes equations with non-Newtonian viscosity is apparently the most appropriate mathematical model for the physics of brain aneurysm blood flow. Due to various factors (discussed in Section 2.2.1), true turbulence in intracranial aneurysms is probably rare, and thus the laminar assumption should hold. Complex and significant temporal variations such as aneurysmal flow reversal[GCOM92] have been observed in pulsatile models, and a transient simulation is necessary to capture these potentially important phenomena. Using three-dimensional models rather than two-dimensional is the only way to capture the effects of secondary flows, which have been found to be significant[NKS84]. Using simpler hydrodynamic models rather than the full Navier-Stokes equations would sacrifice the detailed flow data (e.g. stress concentrations and inflow zones) that will be crucial for surgical planning. The appropriate blood viscosity model is less conclusive, though there was a significant difference in the flow simulation results between Newtonian and non-Newtonian models. Considering this and the viscous properties of blood, it would seem that pursuing accurate non-Newtonian blood models would be most promising.

The object-oriented flow data base was a successful solution to the problem of real time random access to the simulated flow. As more precise data is computed and more precomputed data must be stored (such as isosurfaces) the data base size may exceed the limits of real memory. In this case, a more sophisticated scheme such as predictive caching may be necessary.

2. How useful are computational fluid dynamics techniques for application to solving pulsatile brain aneurysm blood flows?

Computational fluid dynamics software was successfully applied to the brain aneurysm blood flow problem. General purpose commercial CFD packages are available and were used for simulating the developed flow models. Solution for one heartbeat on workstation systems typically took 4 to 8 days. Such a duration is probably unacceptable for clinical use. This problem can most easily be solved by either waiting for faster workstations, or acquiring dedicated time on highly parallel supercomputers.

When selecting flow simulation software, the meshing strategy used is especially important for the brain aneurysm problem. Since the aneurysm geometry can be quite detailed and irregular, using a package that supports tetrahedral meshes is imperative. Furthermore, due to the intensive nature of meshing and remeshing irregular three-dimensional meshes, a package should support automatic meshing and adaptive remeshing. Combined together, these two features allow simulation with minimal operator involvement.

A major difficulty encountered was how to effectively validate the simulated flow results. At issue was not only how to validate, but to what extent (see Section 3.5). Qualitatively, the simulation results were found to match in vivo results (locally acquired cineangiography of swine aneurysms as well as [GSPR92]), in vitro results[LSPR87, NKS84], as well as other numerical simulations (especially the sophisticated models of [LPR93]). Attempts at quantitative validation largely failed due to difficulties in using an endovascular Doppler probe for making precise measurements of flow (as discussed in Section 3.6). If more quantitative validation is required, new validation techniques will need to be developed. Until then, we can have some confidence on the results based on the known validity of the individual techniques involved, and the agreement with other published results.

3. How appropriate are virtual reality interfaces for visualization in a medical setting?

It was found that certain aspects of virtual reality interfaces are more or less appropriate in a medical application. In the case of visualization of brain aneurysm blood flow, a fully immersive interface (e.g. head mounted display) is probably not necessary or even advantageous. The three-dimensional nature of the anatomy and blood flow can be communicated better using a three-dimensional monitor. The advantages are that the physician can still interact with the real world, and can easily switch attention from the data to other external concerns.

An immersive interface allows direct three-dimensional manipulation of virtual objects in the world. A 3D input device (such as a dataglove or 3D mouse) seems natural for positioning objects. Relying on such a device for controlling other parameters can be difficult. Trying to map many lower dimensional controls (e.g. 1D color map range or 2D cutting plane size) all to the same three dimensional input device can result in a cluttered and unintuitive interface. Display of textual or complex system state data is also problematic in an immersive virtual environment. The results of this study suggest that while the three-dimensional, interactive, and direct manipulation components of virtual environments are useful for medical visualization of 3D data, full immersiveness can be a negative factor.

5.3 Contributions

Many contributions were made while investigating the questions discussed above. Due to the interdisciplinary aspects of the research, understanding of some of the problems in both medicine and computer science was increased. The following list outlines some of the major contributions made.

1. Developed and tested modeling techniques for aneurysm geometry and physiology.

- Constructed spline-based surface model from biplanar digital subtraction angiography images.
- Mathematically modeled non-Newtonian viscosity of blood.
- Modeled blood flow using partial differential equations with boundary conditions matched to in vivo aneurysm blood flow.

2. Used computational fluid dynamics to simulate pulsatile laminar non-Newtonian blood flow in rigid lateral wall brain aneurysms.

- Compared Newtonian and non-Newtonian blood flow simulations.
- Performed qualitative and quantitative validation from digital subtraction angiography and Doppler velocimetry in animal models.

3. Designed object-oriented flow database.

- Fast index to spatial data.
- Precomputation for subsequent real-time access.
- Parallel access on symmetric multiprocessor architecture.

4. Designed, implemented, and evaluated prototype virtual environment for medical flow visualization.

- Designed object-oriented framework for easy addition of new virtual tools.
- Enabled and tested hands-free navigation via voice command.
- Implemented and tested directly manipulable embedded tool controls.

5.4 Suggestions for Future Research

This work was a first step towards a vision of a comprehensive clinical tool for assisting radiologists in making crucial decisions. Much more research needs to be done to fulfill this vision.

More realistic models need to be developed to represent not only the detailed shapes of aneurysms, but their elastic properties. The initial models could be statically based on images (perhaps DSA), and subsequently physically based. Along with new mathematical models, new numerical simulation system will be required to solve the coupled system of wall and fluid forces. The preliminary elastic aneurysm blood simulation results of Low et al.[LPR93] suggest that simulating wall elasticity can quantitatively alter flow results.

A surgical simulation system could be developed in several stages, based upon the VR blood flow visualization system. A first step would be to physically simulate the insertion of endovascular coils (or other devices such as balloons). Such a system would have to model the geometric interaction of the microcatheter, guide wire, and coil with the aneurysm and artery walls. It would allow surgical plans to be tested a priori to determine, for example, if a particular coil would fit in a portion of the aneurysm, or how far coils could be packed into the neck without them slipping out into the artery. The second step would be to simulate (off line) the effects of coils on the blood flow. By visualizing the velocity, pressure, and stress fields after coil insertion, a physician could evaluate the probable effectiveness of a particular treatment. Building on this, a third stage of the simulation system would allow real-time viewing of the simulated flow effects of the coils and microcatheter. This would allow true surgical simulation with immediate feedback of disturbed aneurysm blood flow. Fulfilling this long term goal will require many research challenges to be faced in modeling, simulation, and 3D visualization.

Appendix A

Flowfield library source

A.1 types.h

```
#ifndef __types_h
#define __types_h

typedef int nodeindex;
typedef int faceindex;
typedef int cellindex;
typedef float Float;

const int N_of_Buckets = 1000;
const Float EPSILON = 0.0001;

#endif   /* !__types_h */
```

A.2 Position.h

```
#ifndef __Position_h
#define __Position_h

#include <iostream.h>
#include <math.h>
#include "types.h"

/// The spatial position of a point, or a vector from the origin.
/**
Represents the spatial location of a point.  Performs vector algebra
operations.
*/

class Position
{

protected:

  friend istream& operator>>(istream& s, Position& p);
  friend ostream& operator<<(ostream& s, const Position& p);
```

```
  Float X;     // X coordinate.
  Float Y;     // Y coordinate.
  Float Z;     // Z coordinate.

public:

    //  Default constructor sets all coordinates to zero.
  inline Position();

    ///  Explicit constructor takes X, Y, and Z.
  inline Position(const Float& x, const Float& y, const Float& z);

    //  Assignment operator.
  inline Position& operator=(const Position& p);

    //  Copy constructor.
  inline Position(const Position& p);

    //  Destructor.
  inline ~Position();

  /// Returns x-coordinate.
  inline const Float& x(void) const;

  /// Returns y-coordinate.
  inline const Float& y(void) const;

  /// Returns z-coordinate.
  inline const Float& z(void) const;

  /// Dot product of two vectors.
  const Float dot(const Position& pos) const;

  /// Cross product of two vectors.
  const Position cross(const Position& pos) const;

  /// Distance from origin.
  const Float magnitude(void) const;

  /// Normalizes vector so length is 1.
  void normalize(void);

};

/// Combination of position and (possibly fractional) timestep.
class PosTimestep
{
private:

  friend istream& operator>>(istream& s, PosTimestep& pt);
  friend ostream& operator<<(ostream& s, const PosTimestep& pt);

  Position Pos;
  Float Ts;        // timestep
```

```
public:

  PosTimestep();
  /// Construct with position and floating timestep.

  PosTimestep(const Position& pos, const Float& ts);
  /// Construct with position and integer timestep.

  PosTimestep(const Position& pos, const int& ts);

  PosTimestep& operator=(const PosTimestep& pt);

  PosTimestep(const PosTimestep& pt);

  ~PosTimestep();

  /// Return the position.
  const Position& position() const;

  /// Return integer timestep.
  const int timestep() const;

  /// Return floating timestep.
  const Float& ftimestep() const;

};

istream& operator>>(istream& s, PosTimestep& pt);

ostream& operator<<(ostream& s, const PosTimestep& pt);

bool operator==(const PosTimestep& pt1, const PosTimestep& pt2);

  //  Read position from input stream.
istream& operator>>(istream& s, Position& p);

  //  Write position to output stream.
ostream& operator<<(ostream& s, const Position& p);

  // Equality operator == checks whether two positions are nearly equal.
const int operator==(const Position& positionA,
     const Position& positionB);

  // Greater than is undefined, but necessary for Range template
const int operator>(const Position&, const Position&);

  // Less than is undefined, but necessary for Range template
const int operator<(const Position&, const Position&);

  // Greater or equal is undefined, but necessary for Range template
const int operator>=(const Position&, const Position&);

  // Less or equal is undefined, but necessary for Range template
```

```
const int operator<=(const Position&, const Position&);

  /// Add two vectors.
Position operator+(const Position& pA, const Position& pB);

  /// Subtract two vectors.
Position operator-(const Position& pA, const Position& pB);

  /// Scale a vector.
Position operator*(const Float& scale,
   const Position& p);

inline
Position::Position()
{ X = Y = Z = 0.0; };

inline
Position::Position(const Float& x, const Float& y, const Float& z)
{
  X = x;
  Y = y;
  Z = z;
};

inline
Position& Position::operator=(const Position& p)
{
  if (this == &p) return *this;
  X = p.x();
  Y = p.y();
  Z = p.z();
  return *this;
};

inline
Position::Position(const Position& p)
{ *this = p; };

inline
Position::~Position()
{ };

inline
const Float& Position::x(void) const
{ return X; };

inline
const Float& Position::y(void) const
{ return Y; };

inline
const Float& Position::z(void) const
{ return Z; };
```

```
inline
const Position operator-(const Position& p)
{ return Position(-p.x(),-p.y(),-p.z()); };

inline
Position operator*(const Float& scale,
 const Position& p)
{
  return Position(scale*p.x(),
  scale*p.y(),
  scale*p.z());
};

inline
const Float Position::dot(const Position& p) const
{
  return x()*p.x() + y()*p.y() + z()*p.z();
};

inline
Position operator-(const Position& pA,
   const Position& pB)
{
  return Position(pA.x() - pB.x(),
  pA.y() - pB.y(),
  pA.z() - pB.z());
};

inline
const Position& PosTimestep::position() const
{ return Pos; };

inline
const int PosTimestep::timestep() const
{ return (int)Ts; };

inline
const Float& PosTimestep::ftimestep() const
{ return Ts; };

inline
Position operator+(const Position& pA,
   const Position& pB)
{
  return Position(pA.x() + pB.x(), pA.y() + pB.y(), pA.z() + pB.z());
};

#endif     /* !__Position_h */
```

A.3 Position.cc

```
#include "Position.h"
```

```
istream& operator>>(istream& s, Position& p)
{
  return s >> p.X
           >> p.Y
   >> p.Z;
};

ostream& operator<<(ostream& s, const Position& p)
{
  return s << p.x() << " "
           << p.y() << " "
   << p.z() << " ";
};

const int operator==(const Position& positionA,
     const Position& positionB)
{
  return fabs(positionA.x() - positionB.x()) < EPSILON &&
         fabs(positionA.y() - positionB.y()) < EPSILON &&
 fabs(positionA.z() - positionB.z()) < EPSILON;
};

const int operator>(const Position&, const Position&)
{
  cerr << "Error, > not defined for Position.\n";
  return 0;
};

const int operator<(const Position&, const Position&)
{
  cerr << "Error, < not defined for Position.\n";
  return 0;
};

const int operator>=(const Position&, const Position&)
{
  cerr << "Error, >= not defined for Position.\n";
  return 0;
};

const int operator<=(const Position&, const Position&)
{
  cerr << "Error, <= not defined for Position.\n";
  return 0;
};

PosTimestep::PosTimestep()
{};

PosTimestep::PosTimestep(const Position& pos, const int& ts)
: Pos(pos), Ts(ts)
{};

PosTimestep::PosTimestep(const Position& pos, const Float& ts)
```

```
: Pos(pos), Ts(ts)
{};

PosTimestep& PosTimestep::operator=(const PosTimestep& pt)
{
  if (this == &pt) return *this;
  Pos = pt.Pos;
  Ts = pt.Ts;
  return *this;
};

PosTimestep::PosTimestep(const PosTimestep& pt)
{ *this = pt; };

PosTimestep::~PosTimestep()
{};

istream& operator>>(istream& s, PosTimestep& pt)
{
  return s >> pt.Pos
           >> pt.Ts;
};

ostream& operator<<(ostream& s, const PosTimestep& pt)
{
  return s << pt.Pos << " "
           << pt.Ts << " ";
};

bool operator==(const PosTimestep& pt1, const PosTimestep& pt2)
{
  return    (pt1.position() == pt2.position())
         && (fabs(pt1.timestep() - pt2.timestep()) < EPSILON);
};

const Position Position::cross(const Position& p) const
{
  return Position(y()*p.z() - z()*p.y(),
  z()*p.x() - x()*p.z(),
  x()*p.y() - y()*p.x());
};

const Float Position::magnitude(void) const
{
  return sqrt(x()*x() + y()*y() + z()*z());
};

void Position::normalize(void)
{
  Float mag = magnitude();
  if (mag > EPSILON*EPSILON)
    {
      X /= mag;
      Y /= mag;
```

```
      Z /= mag;
    }
  else
    {
      cerr << "error in Position::Normalize, null Position.\n";
    };
};
```

A.4 Quaternion.h

```
#ifndef __Quaternion_h
#define __Quaternion_h

#include <iostream.h>
#include <math.h>
#include "types.h"
#include "Position.h"

/// Spatial orientation or rotation.
/**
Represents spatial orientation.  Responsible for rotational algebra,
rotating points, and inverse rotation.
*/

class Quaternion
{

private:

  friend istream& operator>>(istream& s, Quaternion& q);
  friend ostream& operator<<(ostream& s, const Quaternion& q);

  Float W;     // Twist component.
  Position Axis;

public:

    //  Default constructor sets all coordinates to zero.
  Quaternion();

    ///  Explicit constructor takes 2*rotation w, and rotation axis vector.
  Quaternion(const Float& w, const Position& axis);

    //  Assignment operator.
  Quaternion& operator=(const Quaternion& q);

    //  Copy constructor.
  Quaternion(const Quaternion& q);

    //  Destructor.
  ~Quaternion();

  /// Vector of rotation axis.
```

```
  const Position& axis(void) const;

  /// Returns twist component, 2*rotation angle in radians.
  const Float& w(void) const;

  /// Quaternion with opposite rotation.
  const Quaternion inverse(void) const;

};

  //  Read Quaternion from input stream.
istream& operator>>(istream& s, Quaternion& q);

  //  Write Quaternion to output stream.
ostream& operator<<(ostream& s, const Quaternion& q);

  // Equality operator == checks whether two Quaternions are nearly equal.
const int operator==(const Quaternion& qA,
     const Quaternion& qB);

  // These ops are undefined, but necessary for Range template
const int operator>(const Quaternion&, const Quaternion&);
const int operator<(const Quaternion&, const Quaternion&);
const int operator>=(const Quaternion&, const Quaternion&);
const int operator<=(const Quaternion&, const Quaternion&);

  /// Returns the result of subsequent rotations.
Quaternion operator*(const Quaternion& qA,
     const Quaternion& qB);

  /// Rotates a point about the origin.
Position operator*(const Position& p, const Quaternion& q);

  // defining addition to do subsequent rotations as an analog to
  // translational addition.  This was done so that the DynVar template
  // could use + as the "Delta" operation.
inline
Quaternion operator+(const Quaternion& qA,
     const Quaternion& qB)
{
  return qA*qB;
};

inline
Quaternion::Quaternion()
: Axis(0,0,0)
{
  W = 1;
};

inline
Quaternion::Quaternion(const Quaternion& q)
{ *this = q; };
```

```
inline
Quaternion::~Quaternion()
{ };

inline
const Float& Quaternion::w(void) const
{ return W; };

#endif     /* !__Quaternion_h */
```

A.5 Quaternion.cc

```
#include "Quaternion.h"

Quaternion::Quaternion(const Float& w, const Position& axis)
{
  Float scale = w*w + axis.x()*axis.x()
    + axis.y()*axis.y()
                    + axis.z()*axis.z();
  Float invscale = 1.0/scale;
  W = invscale * w;
  Axis = invscale * axis;
};

const Position& Quaternion::axis(void) const
{
  return Axis;
};

Quaternion& Quaternion::operator=(const Quaternion& q)
{
  if (this == &q) return *this;
  W = q.w();
  Axis = q.axis();
  return *this;
};

const Quaternion Quaternion::inverse(void) const
{
  return Quaternion(-W, Axis);
};

istream& operator>>(istream& s, Quaternion& q)
{
  return s >> q.W
   >> q.Axis;
};

ostream& operator<<(ostream& s, const Quaternion& q)
{
  return s << q.w() << " "
           << q.axis() << " ";
};
```

```
const int operator==(const Quaternion& qA,
     const Quaternion& qB)
{
  return fabs(qA.w() - qB.w()) < EPSILON &&
         fabs(qA.axis().x() - qB.axis().x()) < EPSILON &&
         fabs(qA.axis().y() - qB.axis().y()) < EPSILON &&
 fabs(qA.axis().z() - qB.axis().z()) < EPSILON;
};

const int operator>(const Quaternion&, const Quaternion&)
{
  cerr << "Error, > not defined for Quaternion.\n";
  return 0;
};

const int operator<(const Quaternion&, const Quaternion&)
{
  cerr << "Error, < not defined for Quaternion.\n";
  return 0;
};

const int operator>=(const Quaternion&, const Quaternion&)
{
  cerr << "Error, >= not defined for Quaternion.\n";
  return 0;
};

const int operator<=(const Quaternion&, const Quaternion&)
{
  cerr << "Error, <= not defined for Quaternion.\n";
  return 0;
};

Quaternion operator*(const Quaternion& qA,
     const Quaternion& qB)
{
  return Quaternion(qA.w()*qB.w() - qA.axis().dot(qB.axis()),
      qA.w()*qB.axis() + qB.w()*qA.axis()
    + qA.axis().cross(qB.axis()));
};

Position operator*(const Position& p,
   const Quaternion& q)
{
  Float w = q.w();
  Float x = q.axis().x();
  Float y = q.axis().y();
  Float z = q.axis().z();
  Float m11 = 1 - 2*y*y - 2*z*z;
  Float m22 = 1 - 2*x*x - 2*z*z;
  Float m33 = 1 - 2*x*x - 2*y*y;
  Float m12 = 2*x*y + 2*w*z;
  Float m21 = 2*x*y - 2*w*z;
```

```
  Float m31 = 2*x*z + 2*w*y;
  Float m13 = 2*x*z - 2*w*y;
  Float m23 = 2*y*z + 2*w*x;
  Float m32 = 2*y*z - 2*w*x;
  return Position(p.x()*m11 + p.y()*m21 + p.z()*m31,
  p.x()*m12 + p.y()*m22 + p.z()*m32,
  p.x()*m13 + p.y()*m23 + p.z()*m33);
};
```

A.6 StrainRate.h

```
#ifndef __StrainRate_h
#define __StrainRate_h

#include "Position.h"
class Flowpoint; // fwd dec

class StrainRate
{
private:

  friend istream& operator>>(istream& s, StrainRate& sr);
  friend ostream& operator<<(ostream& s, const StrainRate& sr);
  friend StrainRate operator+(const StrainRate& srA,
         const StrainRate& srB);
  friend StrainRate operator-(const StrainRate& srA,
      const StrainRate& srB);
  friend StrainRate operator*(const Float& mult,
      const StrainRate& sr);

  Position U;   // the derivative vector for U
  Position V;   // the derivative vector for V
  Position W;   // the derivative vector for W
  Position P;   // the derivative vector for P

public:

  StrainRate();

  StrainRate(const Flowpoint& gx, const Flowpoint& gy, const Flowpoint& gz);

  const Flowpoint operator*(Position pos) const;

  Position surfstrainrate(const Position& normal) const;

  const Float absstrainrate(void) const;

  const Float divergence(void) const;

};

StrainRate operator+(const StrainRate& srA,
```

```
     const StrainRate& srB);

StrainRate operator-(const StrainRate& srA,
     const StrainRate& srB);

StrainRate operator*(const Float& mult,
     const StrainRate& sr);

istream& operator>>(istream& s, StrainRate& sr);

ostream& operator<<(ostream& s, const StrainRate& sr);


inline
StrainRate::StrainRate()
: U(0,0,0), V(0,0,0), W(0,0,0), P(0,0,0)
{ };

#endif /* !__StrainRate_h */
```

A.7 Linearfield.h

```
#ifndef __Linearfield_h
#define __Linearfield_h

#include <iostream.h>
#include "Position.h"
#include "StrainRate.h"
#include <assert.h>

class Flowpoint; // fwd declaration

/// A 3D continuous and infinite linear field of flowpoints.
/**
  Represents a single linearly varying field of flow parameters.
  Automatically constructs field from four points, interpolates data at a
  point.
*/

class Linearfield
{

private:

  friend istream& operator>>(istream& s, Linearfield& lf);
  friend ostream& operator<<(ostream& s, const Linearfield& lf);

  Position Zerovelocity;
  Float Zeropressure;
  StrainRate Gradient;

public:
```

```
  //  Default constructor gives an all zero field
  Linearfield();

  /// Construct from four Positions and corresponding Flowpoints.
  Linearfield(const Position pos[4], const Flowpoint fp[4]);

    // Assignment operator
  Linearfield& operator=(const Linearfield& lf);

    //  Copy constructor.
  Linearfield(const Linearfield& lf);

  ~Linearfield();   // Destructor.

  /// Get flowpoint value of field at a position.
  const Flowpoint value(const Position& pos) const;

  const StrainRate& strainrate(void) const;

};

  //  Read linearfield from input stream.
istream& operator>>(istream& s, Linearfield& lf);

  //  Write linearfield to output stream.
ostream& operator<<(ostream& s, const Linearfield& lf);

// These operators are illegal to call, but necessary for vector
const int operator==(const Linearfield&, const Linearfield&);
const int operator>(const Linearfield&, const Linearfield&);
const int operator<(const Linearfield&, const Linearfield&);
const int operator>=(const Linearfield&, const Linearfield&);
const int operator<=(const Linearfield&, const Linearfield&);

inline
const StrainRate& Linearfield::strainrate(void) const
{
  return Gradient;
};

#endif    /* !__Linearfield_h */
```

A.8 Flowpoint.h

```
#ifndef __Flowpoint_h
#define __Flowpoint_h

#include <iostream.h>
#include <math.h>
#include <assert.h>
#include "types.h"
```

```
#include "Quaternion.h"
#include "Linearfield.h"
#include "StrainRate.h"

/// Fluid flow at a single point in space and time.
/**
Represents the velocity and pressure of a fluid at one position and
time.  Also stores strain rate field, and compute planar strain rate.
Computes maximum speed, and speed in a direction.  Performs
vector algebra and scaling.
*/
class Flowpoint
{
private:

  friend istream& operator>>(istream& s, Flowpoint& f);
  friend ostream& operator<<(ostream& s, const Flowpoint& f);
  friend Flowpoint operator*(const Float& mult, const Flowpoint& fp);

  Position Velocity;
  Float Pressure;
  StrainRate VelGrad;

public:

  //  Default constructor sets all Flow to zero.
  Flowpoint();

  ///  Explicit constructor accepts inits for velocity and pressure.
  Flowpoint(const Position& vel, const Float& press);

  void setstrainrate(const StrainRate& sr)
    {
      VelGrad = sr;
    };

  //  Assignment operator
  Flowpoint& operator=(const Flowpoint& fp);

  //  Copy constructor.
  Flowpoint(const Flowpoint& f);

  ~Flowpoint();   // Destructor.

  /// Returns summed velocity and pressure values.
  Flowpoint operator+(const Flowpoint& fp) const;

  /// Returns subtracted velocity and pressure values.
  Flowpoint operator-(const Flowpoint& fp) const;

  /// Velocity of the point, interpreted as a vector from origin to position.
  inline const Position& velocity(void) const;

  /// Pressure value.
```

```
  inline const Float& pressure(void) const;

  /// Absolute speed.
  inline const Float speed(void) const;

  /// Projected speed in direction of q, where q is rotated z vector.
  const Float speed(const Quaternion& q) const;

  /// Surface strain rate vector in plane normal to p vector.
  const Position surfstrainrate(Position normal) const;

  const Float absstrainrate(void) const;
};

//  Read Flowpoint from input stream.
istream& operator>>(istream& s, Flowpoint& fp);

//  Write Flowpoint to output stream.
ostream& operator<<(ostream& s, const Flowpoint& fp);

// Equality operator == checks whether two Flowpoints are nearly equal.
int operator==(const Flowpoint& fpA,
       const Flowpoint& fpB);

// Greater than is undefined, but necessary for Range template
const int operator>(const Flowpoint&, const Flowpoint&);

// Less than is undefined, but necessary for Range template
const int operator<(const Flowpoint&, const Flowpoint&);

// Greater or equal is undefined, but necessary for Range template
const int operator>=(const Flowpoint&, const Flowpoint&);

// Less or equal is undefined, but necessary for Range template
const int operator<=(const Flowpoint&, const Flowpoint&);

// Multiply operator, scales each value
Flowpoint operator*(const Float& mult, const Flowpoint& fp);

inline
Flowpoint::Flowpoint(const Flowpoint& fp)
{ *this = fp; };

inline
const Position& Flowpoint::velocity(void) const
{ return Velocity; };

inline
const Float& Flowpoint::pressure(void) const
{ return Pressure; };

inline
const Float Flowpoint::speed(void) const
{ return Velocity.magnitude(); };
```

```
inline
Flowpoint operator*(const Float& mult, const Flowpoint& fp)
{
  Flowpoint tmpfp(mult * fp.velocity(),
  mult * fp.pressure());
  tmpfp.VelGrad = mult * fp.VelGrad;
  return tmpfp;
};

#endif    /* !__Flowpoint_h */
```

A.9 StrainRate.cc

```
#include "StrainRate.h"
#include "Flowpoint.h"

StrainRate::StrainRate(const Flowpoint& gx,
       const Flowpoint& gy,
       const Flowpoint& gz)
{
  U = Position(gx.velocity().x(), gy.velocity().x(), gz.velocity().x());
  V = Position(gx.velocity().y(), gy.velocity().y(), gz.velocity().y());
  W = Position(gx.velocity().z(), gy.velocity().z(), gz.velocity().z());
  P = Position(gx.pressure(), gy.pressure(), gz.pressure());
};


Position StrainRate::surfstrainrate(const Position& normal) const
{
  return Position(U.dot(normal), V.dot(normal), W.dot(normal));
};

const Float StrainRate::absstrainrate(void) const
{
  const Float exy = 0.5*(U.y() + V.x());
  const Float eyz = 0.5*(V.z() + W.y());
  const Float exz = 0.5*(U.z() + W.x());

  return 2*sqrt(exy*exy + eyz*eyz + exz*exz);
};

const Float StrainRate::divergence(void) const
{
  return U.x() + V.y() + W.z();
};

const Flowpoint StrainRate::operator*(Position pos) const
{
  Flowpoint tmpfp(Position(U.dot(pos), V.dot(pos), W.dot(pos)), P.dot(pos));
  tmpfp.setstrainrate(*this);
  return tmpfp;
```

```
};

istream& operator>>(istream& s, StrainRate& sr)
{
  return s >> sr.U >> sr.V >> sr.W >> sr.P;
};

ostream& operator<<(ostream& s, const StrainRate& sr)
{
  return s << sr.U << " " << sr.V << " "
           << sr.W << " " << sr.P << "\n";
};

StrainRate operator+(const StrainRate& srA,
     const StrainRate& srB)
{
  StrainRate sr;
  sr.U = srA.U + srB.U;
  sr.V = srA.V + srB.V;
  sr.W = srA.W + srB.W;
  sr.P = srA.P + srB.P;
  return sr;
};

StrainRate operator-(const StrainRate& srA,
     const StrainRate& srB)
{
  StrainRate sr;
  sr.U = srA.U - srB.U;
  sr.V = srA.V - srB.V;
  sr.W = srA.W - srB.W;
  sr.P = srA.P - srB.P;
  return sr;
};

StrainRate operator*(const Float& mult,
     const StrainRate& sr)
{
  StrainRate srtmp;
  srtmp.U = mult*sr.U;
  srtmp.V = mult*sr.V;
  srtmp.W = mult*sr.W;
  srtmp.P = mult*sr.P;
  return srtmp;
};
```

A.10 Linearfield.cc

```
#include "Linearfield.h"
#include "Flowpoint.h"

Linearfield::Linearfield()
{
```

```
  Zerovelocity = Position(0.0, 0.0, 0.0);
  Zeropressure = 0.0;
};

Linearfield& Linearfield::operator=(const Linearfield& lf)
{
  if (this == &lf) return *this;
  Zerovelocity = lf.Zerovelocity;
  Zeropressure = lf.Zeropressure;
  Gradient = lf.Gradient;
  return *this;
};

Linearfield::Linearfield(const Linearfield& lf)
{ *this = lf; };

Linearfield::~Linearfield()
{ };

istream& operator>>(istream& s, Linearfield& lf)
{
  return s >> lf.Zerovelocity
           >> lf.Zeropressure
           >> lf.Gradient;
};

ostream& operator<<(ostream& s, const Linearfield& lf)
{
  return s << lf.Zerovelocity << "\n"
           << lf.Zeropressure << "\n"
           << lf.Gradient << "\n";
};


// definition of the constructor - lots of ugly maple generated code.
Linearfield::Linearfield(const Position pos[4],
                         const Flowpoint fp[4])
{
  /*
     This member must solve the gradient and zero value
     of the field given four sample points in the field.
     Here is the Maple code which generates the C code:

     # start maple code
     with(linalg);
     readlib(C);
     F := matrix([[fp_0_u, fp_1_u, fp_2_u, fp_3_u],
                  [fp_0_v, fp_1_v, fp_2_v, fp_3_v],
                  [fp_0_w, fp_1_w, fp_2_w, fp_3_w],
                  [fp_0_p, fp_1_p, fp_2_p, fp_3_p]]);
     X := matrix([[pos_0_x, pos_1_x, pos_2_x, pos_3_x],
                  [pos_0_y, pos_1_y, pos_2_y, pos_3_y],
                  [pos_0_z, pos_1_z, pos_2_z, pos_3_z],
                  [1,       1,       1,       1      ]]);
```

```
   GF0 := multiply(F, inverse(X));
   C(GF0, optimized);
   quit;
   # end maple code

   The resulting code was cleaned up a little.  I macroed the
   pos_1_x form into pos[1].x()    -the latter form screwed
   up Maple.

 */

// The temporary variables - from Maple´s optimization
Float t1 = pos[2].y()*pos[3].z();
Float t2 = pos[1].x()*t1;
Float t5 = pos[3].y()*pos[2].z();
Float t6 = pos[1].x()*t5;
Float t8 = pos[2].x()*pos[3].z();
Float t9 = pos[1].y()*t8;
Float t11 = pos[3].x()*pos[2].z();
Float t12 = pos[1].y()*t11;
Float t15 = pos[2].x()*pos[3].y();
Float t16 = pos[1].z()*t15;
Float t19 = pos[3].x()*pos[2].y();
Float t20 = pos[1].z()*t19;
Float t23 = pos[0].x()*t1;
Float t25 = pos[0].x()*t5;
Float t28 = pos[0].y()*t8;
Float t31 = pos[0].y()*t11;
Float t33 = pos[0].z()*t15;
Float t35 = pos[0].z()*t19;
Float t40 = pos[1].y()*pos[3].z();
Float t41 = pos[0].x()*t40;
Float t44 = pos[1].z()*pos[3].y();
Float t45 = pos[0].x()*t44;
Float t47 = pos[1].x()*pos[3].z();
Float t48 = pos[0].y()*t47;
Float t50 = pos[1].z()*pos[3].x();
Float t51 = pos[0].y()*t50;
Float t54 = pos[1].x()*pos[3].y();
Float t55 = pos[0].z()*t54;
Float t58 = pos[1].y()*pos[3].x();
Float t59 = pos[0].z()*t58;
Float t62 = pos[1].y()*pos[2].z();
Float t63 = pos[0].x()*t62;
Float t65 = pos[1].z()*pos[2].y();
Float t66 = pos[0].x()*t65;
Float t69 = pos[1].x()*pos[2].z();
Float t70 = pos[0].y()*t69;
Float t73 = pos[1].z()*pos[2].x();
Float t74 = pos[0].y()*t73;
Float t76 = pos[1].x()*pos[2].y();
Float t77 = pos[0].z()*t76;
Float t79 = pos[1].y()*pos[2].x();
Float t80 = pos[0].z()*t79;
```

```
Float t105 = 1/(t63-t41-t66+t45+t23-t25
 -t70+t48+t74-t51-t28+t31+t77-t55-t80+t59
 +t33-t35-t2+t6+t9-t12-t16+t20);
Float t117 = pos[0].y()*pos[2].z();
Float t120 = pos[0].y()*pos[3].z();
Float t122 = pos[0].z()*pos[2].y();
Float t124 = pos[0].z()*pos[3].y();
Float t132 = pos[0].y()*pos[1].z();
Float t136 = pos[0].z()*pos[1].y();
Float t167 = pos[0].x()*pos[2].z();
Float t169 = pos[0].x()*pos[3].z();
Float t172 = pos[0].z()*pos[2].x();
Float t175 = pos[0].z()*pos[3].x();
Float t182 = pos[0].x()*pos[1].z();
Float t186 = pos[0].z()*pos[1].x();
Float t261 = pos[0].x()*pos[2].y();
Float t263 = pos[0].x()*pos[3].y();
Float t266 = pos[0].y()*pos[2].x();
Float t269 = pos[0].y()*pos[3].x();
Float t276 = pos[0].x()*pos[1].y();
Float t280 = pos[0].y()*pos[1].x();

// This is the solution matrix, the first three columns are
// the gradient, the last column is the zero position Flowpoint.
// I use a 5x5 array here rather than a 4x4 array to work around
// maple´s 1 based array indices.
Float GF0[5][5];

GF0[3][4] = (-fp[0].velocity().z()*t2+fp[0].velocity().z()*t6
     +fp[0].velocity().z()*t9-fp[0].velocity().z()*t12
     -fp[0].velocity().z()*t16+fp[0].velocity().z()*t20
     +fp[1].velocity().z()*t23-fp[1].velocity().z()*t25
     -fp[1].velocity().z()*t28+fp[1].velocity().z()*t31
     +fp[1].velocity().z()*t33-fp[1].velocity().z()*t35
     -fp[2].velocity().z()*t41+fp[2].velocity().z()*t45
     +fp[2].velocity().z()*t48-fp[2].velocity().z()*t51
     -fp[2].velocity().z()*t55+fp[2].velocity().z()*t59
     +fp[3].velocity().z()*t63-fp[3].velocity().z()*t66
     -fp[3].velocity().z()*t70+fp[3].velocity().z()*t74
     +fp[3].velocity().z()*t77-fp[3].velocity().z()*t80)*t105;

GF0[2][1] = (fp[0].velocity().y()*t62-fp[0].velocity().y()*t40
     -fp[0].velocity().y()*t65+fp[0].velocity().y()*t44
     +fp[0].velocity().y()*t1-fp[0].velocity().y()*t5
     -fp[1].velocity().y()*t117+fp[1].velocity().y()*t120
     +fp[1].velocity().y()*t122-fp[1].velocity().y()*t124
     -fp[1].velocity().y()*t1+fp[1].velocity().y()*t5
     +fp[2].velocity().y()*t132-fp[2].velocity().y()*t120
     -fp[2].velocity().y()*t136+fp[2].velocity().y()*t124
     +fp[2].velocity().y()*t40-fp[2].velocity().y()*t44
     -fp[3].velocity().y()*t132+fp[3].velocity().y()*t117
     +fp[3].velocity().y()*t136-fp[3].velocity().y()*t122
     -fp[3].velocity().y()*t62+fp[3].velocity().y()*t65)*t105;
```

```
  GF0[2][2] = (-fp[0].velocity().y()*t69+fp[0].velocity().y()*t47
       +fp[0].velocity().y()*t73-fp[0].velocity().y()*t50
       -fp[0].velocity().y()*t8+fp[0].velocity().y()*t11
       +fp[1].velocity().y()*t167-fp[1].velocity().y()*t169
       -fp[1].velocity().y()*t172+fp[1].velocity().y()*t175
       +fp[1].velocity().y()*t8-fp[1].velocity().y()*t11
       -fp[2].velocity().y()*t182+fp[2].velocity().y()*t169
       +fp[2].velocity().y()*t186-fp[2].velocity().y()*t175
       -fp[2].velocity().y()*t47+fp[2].velocity().y()*t50
       +fp[3].velocity().y()*t182-fp[3].velocity().y()*t167
       -fp[3].velocity().y()*t186+fp[3].velocity().y()*t172
       +fp[3].velocity().y()*t69-fp[3].velocity().y()*t73)*t105;

  GF0[4][1] = (fp[0].pressure()*t62-fp[0].pressure()*t40
       -fp[0].pressure()*t65+fp[0].pressure()*t44
       +fp[0].pressure()*t1-fp[0].pressure()*t5
       -fp[1].pressure()*t117+fp[1].pressure()*t120
       +fp[1].pressure()*t122-fp[1].pressure()*t124
       -fp[1].pressure()*t1+fp[1].pressure()*t5
       +fp[2].pressure()*t132-fp[2].pressure()*t120
       -fp[2].pressure()*t136+fp[2].pressure()*t124
       +fp[2].pressure()*t40-fp[2].pressure()*t44
       -fp[3].pressure()*t132+fp[3].pressure()*t117
       +fp[3].pressure()*t136-fp[3].pressure()*t122
       -fp[3].pressure()*t62+fp[3].pressure()*t65)*t105;

  GF0[2][3] = -(-fp[0].velocity().y()*t76+fp[0].velocity().y()*t54
+fp[0].velocity().y()*t79-fp[0].velocity().y()*t58
-fp[0].velocity().y()*t15+fp[0].velocity().y()*t19
+fp[1].velocity().y()*t261-fp[1].velocity().y()*t263
-fp[1].velocity().y()*t266+fp[1].velocity().y()*t269
+fp[1].velocity().y()*t15-fp[1].velocity().y()*t19
-fp[2].velocity().y()*t276+fp[2].velocity().y()*t263
+fp[2].velocity().y()*t280-fp[2].velocity().y()*t269
-fp[2].velocity().y()*t54+fp[2].velocity().y()*t58
+fp[3].velocity().y()*t276-fp[3].velocity().y()*t261
-fp[3].velocity().y()*t280+fp[3].velocity().y()*t266
+fp[3].velocity().y()*t76-fp[3].velocity().y()*t79)*t105;

  GF0[4][2] = (-fp[0].pressure()*t69+fp[0].pressure()*t47
       +fp[0].pressure()*t73-fp[0].pressure()*t50
       -fp[0].pressure()*t8+fp[0].pressure()*t11
       +fp[1].pressure()*t167-fp[1].pressure()*t169
       -fp[1].pressure()*t172+fp[1].pressure()*t175
       +fp[1].pressure()*t8-fp[1].pressure()*t11
       -fp[2].pressure()*t182+fp[2].pressure()*t169
       +fp[2].pressure()*t186-fp[2].pressure()*t175
       -fp[2].pressure()*t47+fp[2].pressure()*t50
       +fp[3].pressure()*t182-fp[3].pressure()*t167
       -fp[3].pressure()*t186+fp[3].pressure()*t172
       +fp[3].pressure()*t69-fp[3].pressure()*t73)*t105;

  GF0[2][4] = (-fp[0].velocity().y()*t2+fp[0].velocity().y()*t6
       +fp[0].velocity().y()*t9-fp[0].velocity().y()*t12
```

```
      -fp[0].velocity().y()*t16+fp[0].velocity().y()*t20
      +fp[1].velocity().y()*t23-fp[1].velocity().y()*t25
      -fp[1].velocity().y()*t28+fp[1].velocity().y()*t31
      +fp[1].velocity().y()*t33-fp[1].velocity().y()*t35
      -fp[2].velocity().y()*t41+fp[2].velocity().y()*t45
      +fp[2].velocity().y()*t48-fp[2].velocity().y()*t51
      -fp[2].velocity().y()*t55+fp[2].velocity().y()*t59
      +fp[3].velocity().y()*t63-fp[3].velocity().y()*t66
      -fp[3].velocity().y()*t70+fp[3].velocity().y()*t74
      +fp[3].velocity().y()*t77-fp[3].velocity().y()*t80)*t105;

  GF0[4][3] = -(-fp[0].pressure()*t76+fp[0].pressure()*t54
+fp[0].pressure()*t79-fp[0].pressure()*t58
-fp[0].pressure()*t15+fp[0].pressure()*t19
+fp[1].pressure()*t261-fp[1].pressure()*t263
-fp[1].pressure()*t266+fp[1].pressure()*t269
+fp[1].pressure()*t15-fp[1].pressure()*t19
-fp[2].pressure()*t276+fp[2].pressure()*t263
+fp[2].pressure()*t280-fp[2].pressure()*t269
-fp[2].pressure()*t54+fp[2].pressure()*t58
+fp[3].pressure()*t276-fp[3].pressure()*t261
-fp[3].pressure()*t280+fp[3].pressure()*t266
+fp[3].pressure()*t76-fp[3].pressure()*t79)*t105;

  GF0[1][1] = (fp[0].velocity().x()*t62-fp[0].velocity().x()*t40
      -fp[0].velocity().x()*t65+fp[0].velocity().x()*t44
      +fp[0].velocity().x()*t1-fp[0].velocity().x()*t5
      -fp[1].velocity().x()*t117+fp[1].velocity().x()*t120
      +fp[1].velocity().x()*t122-fp[1].velocity().x()*t124
      -fp[1].velocity().x()*t1+fp[1].velocity().x()*t5
      +fp[2].velocity().x()*t132-fp[2].velocity().x()*t120
      -fp[2].velocity().x()*t136+fp[2].velocity().x()*t124
      +fp[2].velocity().x()*t40-fp[2].velocity().x()*t44
      -fp[3].velocity().x()*t132+fp[3].velocity().x()*t117
      +fp[3].velocity().x()*t136-fp[3].velocity().x()*t122
      -fp[3].velocity().x()*t62+fp[3].velocity().x()*t65)*t105;

  GF0[1][2] = (-fp[0].velocity().x()*t69+fp[0].velocity().x()*t47
      +fp[0].velocity().x()*t73-fp[0].velocity().x()*t50
      -fp[0].velocity().x()*t8+fp[0].velocity().x()*t11
      +fp[1].velocity().x()*t167-fp[1].velocity().x()*t169
      -fp[1].velocity().x()*t172+fp[1].velocity().x()*t175
      +fp[1].velocity().x()*t8-fp[1].velocity().x()*t11
      -fp[2].velocity().x()*t182+fp[2].velocity().x()*t169
      +fp[2].velocity().x()*t186-fp[2].velocity().x()*t175
      -fp[2].velocity().x()*t47+fp[2].velocity().x()*t50
      +fp[3].velocity().x()*t182-fp[3].velocity().x()*t167
      -fp[3].velocity().x()*t186+fp[3].velocity().x()*t172
      +fp[3].velocity().x()*t69-fp[3].velocity().x()*t73)*t105;

  GF0[3][1] = (fp[0].velocity().z()*t62-fp[0].velocity().z()*t40
      -fp[0].velocity().z()*t65+fp[0].velocity().z()*t44
      +fp[0].velocity().z()*t1-fp[0].velocity().z()*t5
      -fp[1].velocity().z()*t117+fp[1].velocity().z()*t120
```

```
      +fp[1].velocity().z()*t122-fp[1].velocity().z()*t124
      -fp[1].velocity().z()*t1+fp[1].velocity().z()*t5
      +fp[2].velocity().z()*t132-fp[2].velocity().z()*t120
      -fp[2].velocity().z()*t136+fp[2].velocity().z()*t124
      +fp[2].velocity().z()*t40-fp[2].velocity().z()*t44
      -fp[3].velocity().z()*t132+fp[3].velocity().z()*t117
      +fp[3].velocity().z()*t136-fp[3].velocity().z()*t122
      -fp[3].velocity().z()*t62+fp[3].velocity().z()*t65)*t105;

  GF0[1][3] = -(-fp[0].velocity().x()*t76+fp[0].velocity().x()*t54
+fp[0].velocity().x()*t79-fp[0].velocity().x()*t58
-fp[0].velocity().x()*t15+fp[0].velocity().x()*t19
+fp[1].velocity().x()*t261-fp[1].velocity().x()*t263
-fp[1].velocity().x()*t266+fp[1].velocity().x()*t269
+fp[1].velocity().x()*t15-fp[1].velocity().x()*t19
-fp[2].velocity().x()*t276+fp[2].velocity().x()*t263
+fp[2].velocity().x()*t280-fp[2].velocity().x()*t269
-fp[2].velocity().x()*t54+fp[2].velocity().x()*t58
+fp[3].velocity().x()*t276-fp[3].velocity().x()*t261
-fp[3].velocity().x()*t280+fp[3].velocity().x()*t266
+fp[3].velocity().x()*t76-fp[3].velocity().x()*t79)*t105;

  GF0[3][2] = (-fp[0].velocity().z()*t69+fp[0].velocity().z()*t47
      +fp[0].velocity().z()*t73-fp[0].velocity().z()*t50
      -fp[0].velocity().z()*t8+fp[0].velocity().z()*t11
      +fp[1].velocity().z()*t167-fp[1].velocity().z()
      *t169-fp[1].velocity().z()*t172+fp[1].velocity().z()*t175
      +fp[1].velocity().z()*t8-fp[1].velocity().z()*t11
      -fp[2].velocity().z()*t182+fp[2].velocity().z()*t169
      +fp[2].velocity().z()*t186-fp[2].velocity().z()*t175
      -fp[2].velocity().z()*t47+fp[2].velocity().z()*t50
      +fp[3].velocity().z()*t182-fp[3].velocity().z()*t167
      -fp[3].velocity().z()*t186+fp[3].velocity().z()*t172
      +fp[3].velocity().z()*t69-fp[3].velocity().z()*t73)*t105;

  GF0[1][4] = (-fp[0].velocity().x()*t2+fp[0].velocity().x()*t6
      +fp[0].velocity().x()*t9-fp[0].velocity().x()*t12
      -fp[0].velocity().x()*t16+fp[0].velocity().x()*t20
      +fp[1].velocity().x()*t23-fp[1].velocity().x()*t25
      -fp[1].velocity().x()*t28+fp[1].velocity().x()*t31
      +fp[1].velocity().x()*t33-fp[1].velocity().x()*t35
      -fp[2].velocity().x()*t41+fp[2].velocity().x()*t45
      +fp[2].velocity().x()*t48-fp[2].velocity().x()*t51
      -fp[2].velocity().x()*t55+fp[2].velocity().x()*t59
      +fp[3].velocity().x()*t63-fp[3].velocity().x()*t66
      -fp[3].velocity().x()*t70+fp[3].velocity().x()*t74
      +fp[3].velocity().x()*t77-fp[3].velocity().x()*t80)*t105;

  GF0[4][4] = (-fp[0].pressure()*t2+fp[0].pressure()*t6
      +fp[0].pressure()*t9-fp[0].pressure()*t12
      -fp[0].pressure()*t16+fp[0].pressure()*t20
      +fp[1].pressure()*t23-fp[1].pressure()*t25
      -fp[1].pressure()*t28+fp[1].pressure()*t31
      +fp[1].pressure()*t33-fp[1].pressure()*t35
```

```
       -fp[2].pressure()*t41+fp[2].pressure()*t45
       +fp[2].pressure()*t48-fp[2].pressure()*t51
       -fp[2].pressure()*t55+fp[2].pressure()*t59
       +fp[3].pressure()*t63-fp[3].pressure()*t66
       -fp[3].pressure()*t70+fp[3].pressure()*t74
       +fp[3].pressure()*t77-fp[3].pressure()*t80)*t105;

  GF0[3][3] = -(-fp[0].velocity().z()*t76+fp[0].velocity().z()*t54
+fp[0].velocity().z()*t79-fp[0].velocity().z()*t58
-fp[0].velocity().z()*t15+fp[0].velocity().z()*t19
+fp[1].velocity().z()*t261-fp[1].velocity().z()*t263
-fp[1].velocity().z()*t266+fp[1].velocity().z()*t269
+fp[1].velocity().z()*t15-fp[1].velocity().z()*t19
-fp[2].velocity().z()*t276+fp[2].velocity().z()*t263
+fp[2].velocity().z()*t280-fp[2].velocity().z()*t269
-fp[2].velocity().z()*t54+fp[2].velocity().z()*t58
+fp[3].velocity().z()*t276-fp[3].velocity().z()*t261
-fp[3].velocity().z()*t280+fp[3].velocity().z()*t266
+fp[3].velocity().z()*t76-fp[3].velocity().z()*t79)*t105;

  // Finally, we build the object using the GF0 array.

  Zerovelocity = Position(GF0[1][4], GF0[2][4], GF0[3][4]);
  Zeropressure = GF0[4][4];
  Position vec0(GF0[1][1], GF0[2][1], GF0[3][1]);
  Position vec1(GF0[1][2], GF0[2][2], GF0[3][2]);
  Position vec2(GF0[1][3], GF0[2][3], GF0[3][3]);
  Flowpoint grad0(vec0, GF0[4][1]);
  Flowpoint grad1(vec1, GF0[4][2]);
  Flowpoint grad2(vec2, GF0[4][3]);
  Gradient = StrainRate(grad0, grad1, grad2);

};

const int operator==(const Linearfield&, const Linearfield&)
{
  cerr << "Error, == not defined for Linearfield.\n";
  return 0;
};

const int operator>(const Linearfield&, const Linearfield&)
{
  cerr << "Error, > not defined for Linearfield.\n";
  return 0;
};

const int operator<(const Linearfield&, const Linearfield&)
{
  cerr << "Error, < not defined for Linearfield.\n";
  return 0;
};

const int operator>=(const Linearfield&, const Linearfield&)
{
```

```
  cerr << "Error, >= not defined for Linearfield.\n";
  return 0;
};

const int operator<=(const Linearfield&, const Linearfield&)
{
  cerr << "Error, <= not defined for Linearfield.\n";
  return 0;
};

const Flowpoint Linearfield::value(const Position& pos) const
{
  Flowpoint fptmp = Gradient*pos + Flowpoint(Zerovelocity, Zeropressure);
  fptmp.setstrainrate(Gradient);
  return fptmp;
};
```

A.11 Flowpoint.cc

```
#include "Flowpoint.h"

Flowpoint::Flowpoint(const Position& vel, const Float& press)
: Velocity(vel), Pressure(press), VelGrad()
{ };

Flowpoint& Flowpoint::operator=(const Flowpoint& fp)
{
  if (this == &fp) return *this;
  Velocity = fp.velocity();
  Pressure = fp.pressure();
  VelGrad = fp.VelGrad;
  return *this;
};

Flowpoint Flowpoint::operator+(const Flowpoint& fp) const
{
  Flowpoint tmpfp(velocity()+fp.velocity(), pressure()+fp.pressure());
  tmpfp.VelGrad = VelGrad + fp.VelGrad;
  return tmpfp;
};

Flowpoint Flowpoint::operator-(const Flowpoint& fp) const
{
  Flowpoint tmpfp(velocity()-fp.velocity(),
  pressure()-fp.pressure());
  tmpfp.VelGrad = VelGrad - fp.VelGrad;
  return tmpfp;
};

const Float Flowpoint::speed(const Quaternion& q) const
{
  return  (Velocity*q.inverse()).z();
};
```

```
const Position Flowpoint::surfstrainrate(Position normal) const
{
  normal.normalize();
  // project velocity gradients onto normal
  return VelGrad.surfstrainrate(normal);
};

const Float Flowpoint::absstrainrate(void) const
{
  return VelGrad.absstrainrate();
};

istream& operator>>(istream& s, Flowpoint& fp)
{
  Position tmpv;
  Float tmpp;
  s >> tmpv
    >> tmpp;
  fp = Flowpoint(tmpv, tmpp);
  return s;
};

ostream& operator<<(ostream& s, const Flowpoint& fp)
{
  return s << fp.Velocity << " "
           << fp.Pressure << " ";
};

int operator==(const Flowpoint& fpA,
       const Flowpoint& fpB)
{
  return fabs(fpA.velocity().x() - fpB.velocity().x()) < EPSILON &&
         fabs(fpA.velocity().y() - fpB.velocity().y()) < EPSILON &&
         fabs(fpA.velocity().z() - fpB.velocity().z()) < EPSILON &&
         fabs(fpA.pressure() - fpB.pressure()) < EPSILON;
};

const int operator>(const Flowpoint&, const Flowpoint&)
{
  cerr << "Error, > not defined for Flowpoint.\n";
  return 0;
};

const int operator<(const Flowpoint&, const Flowpoint&)
{
  cerr << "Error, < not defined for Flowpoint.\n";
  return 0;
};

const int operator>=(const Flowpoint&, const Flowpoint&)
{
  cerr << "Error, >= not defined for Flowpoint.\n";
  return 0;
```

```
};

const int operator<=(const Flowpoint&, const Flowpoint&)
{
  cerr << "Error, <= not defined for Flowpoint.\n";
  return 0;
};

Flowpoint::~Flowpoint()
{ };

Flowpoint::Flowpoint()
: Velocity(0,0,0), Pressure(0), VelGrad()
{ };
```

A.12 Node.h

```
#ifndef __Node_h
#define __Node_h

#include <iostream.h>

#include <vector.h>

#include "Position.h"
#include "Flowpoint.h"

/// A point in a mesh and its flow over time.
/**
  Represents flow at one point over a series of times.  Control
  flowpoints at a mesh point.
*/

class Node
{

private:

  friend istream& operator>>(istream& s, Node& n);
  friend ostream& operator<<(ostream& s, const Node& n);

  Position Pos;          // The spatial location of the node.
  vector<Flowpoint> Fp; // A vector of Flowpoints, one for each timestep.

public:

    //  Default constructor gives you no Flowpoints.
  Node();

    ///  Explicit constructor takes position.
  Node(const Position& pos);

    //  Assignment constructor.
```

```
  Node& operator=(const Node& n);

    //  Copy constructor.
  Node(const Node& n);

    //  Destructor.
  ~Node();

    ///  The position of the node.
  inline const Position& pos(void) const;

    ///  Vector of Flowpoints, one for each timestep.
  inline const vector<Flowpoint>& fptab(void) const;

    ///  Number of Flowpoints in node (number of timesteps.)
  const int num_fp(void) const;

    ///  Returns Flowpoint for timestep i.
  const Flowpoint& fp(const int i) const;

    ///  Sets Flowpoint for timestep i.
  Flowpoint& setfp(const int i);

    ///  Add a new Flowpoint to the node.
  void add_fp(const Flowpoint& fp);

};

  //  Read node from input stream.
istream& operator>>(istream& s, Node& n);

  //  Write node to output stream.
ostream& operator<<(ostream& s, const Node& n);

  //  Equality operator == checks whether two nodes are equal by
  //  checking that their positions are equal.
int operator==(const Node& nodeA,
      const Node& nodeB);

inline
Node::Node()
{ };

inline
Node::Node(const Position& pos)
{ Pos = pos; };

inline
Node& Node::operator=(const Node& n)
{
  if (this == &n) return *this;
  Pos = n.pos();
  Fp = n.fptab();
  return *this;
```

```
};

inline
Node::Node(const Node& n)
{ *this = n; };

inline
Node::~Node()
{ };

inline
const Position& Node::pos(void) const
{ return Pos; };

inline
const vector<Flowpoint>& Node::fptab(void) const
{ return Fp; };

inline
const int Node::num_fp(void) const
{ return Fp.size(); };

inline
const Flowpoint& Node::fp(const int i) const
{ return Fp[i]; };

inline
Flowpoint& Node::setfp(const int i)
{ return Fp[i]; };

inline
void Node::add_fp(const Flowpoint& fp)
{ Fp.push_back(fp); };

#endif    /* !__Node_h */
```

A.13 Node.cc

```
#include "Node.h"

istream& operator>>(istream& s, Node& n)
{
  n.Node::~Node();
  s >> n.Pos;
  int size;
  s >> size;
  n.Fp.reserve(size);
  Flowpoint fp;
  for (int i = 0; i < size; i++)
    {
      s >> fp;
      n.add_fp(fp);
    };
```

```
  return s;
};

ostream& operator<<(ostream& s, const Node& n)
{
  s << n.pos() << "\n";
  int size = n.num_fp();
  s << size << "\n";
  for (int i = 0; i < size; i++)
    {
      s << " " << n.fp(i) << "\n";
    };
  return s << "\n";
};

int operator==(const Node& nodeA,
const Node& nodeB)
{ return nodeA.pos() == nodeB.pos(); };
```

A.14 Face.h

```
#ifndef __Face_h
#define __Face_h

#include <iostream.h>
#include <math.h>
#include "types.h"

#include <vector.h>

#include "Position.h"
#include "Node.h"

class Facep;

/// One side of a tetrahedral cell.
/**
  Represents a connected triangle in space with an inside and outside.
  Handles traversal to an adjacent cell, reporting of external faces,
  automatic connection to other faces, determines if a position is
  inside or outside a cell.
*/

class Face
{
private:

  friend istream& operator>>(istream& s, Face& f);
  friend ostream& operator<<(ostream& s, const Face& f);
  friend int hfunc(const Facep& f);

  nodeindex Ni[3];          // the three nodeindices of the face.
  cellindex Source_cell;    // the cell this face belongs to.
```

```
  faceindex Source_face;    // the number of this face in the cell.
  cellindex Adj_cell;       // the index of the adjacent cell where
                            // -1 is an external face.
  int Orient;               // 0 means right-handed, 1 means left-handed
  Position Normal;          // the normal 3-vector
  Float D;                  // the distance from the plane to the origin

public:

    // Default constructor:
  Face();

    /// Explicit constructor.
  Face(const nodeindex ni0, const nodeindex ni1, const nodeindex ni2,
       cellindex source_cell, faceindex source_face,
       const vector<Node>& nt);

    // Destructor:
  ~Face();

    // Assignment operator:
  Face& operator=(const Face& f);

    // Copy Constructor:
  Face(const Face& f);

  /// Returns the cellindex of the adjacent cell.
  const cellindex adj_cell(void) const;

  /// Returns the cellindex of the owning cell
  const cellindex this_cell(void) const;

  /// Returns the ith node index
  const nodeindex ni(const int i) const;

  /// Returns whether this is an exterior face or not.
  const bool exterior(void) const;

  /// Checks whether a target face is the same, and attaches the two if so.
  const bool check(Face& target);

  /// Returns whether a position is on the inward normal side of the face.
  inline const bool inside(const Position& pos) const;

  /// The outward normal vector.
  const Position normal(void) const
    {
      return Normal;
    };

};

istream& operator>>(istream& s, Face& f);
```

```
ostream& operator<<(ostream& s, const Face& f);

const int operator==(const Face& faceA, const Face& faceB);

const int operator>(const Face&, const Face&);

const int operator<(const Face&, const Face&);

const int operator>=(const Face&, const Face&);

const int operator<=(const Face&, const Face&);

int hfunc(const Facep& f);

inline
const bool Face::inside(const Position& pos) const
{
  return (Normal.dot(pos) >= D - EPSILON*EPSILON);
};

class Facep
{
private:
  Face* Ptr;

public:

  inline Facep() { Ptr = 0; };

  inline Facep(Face* ptr) { Ptr = ptr; };

  inline Facep(Face& face) { Ptr = &face; };

  inline Face* ptr() const { return Ptr; };

  inline Face face() { return *Ptr; };

};

#endif  /* !__Face_h */
```

A.15 Face.cc

```
#include "Face.h"

Face::Face()
: Normal(0,0,0), D(0)
{
  for (int i = 0; i < 3; i++) Ni[i] = -1;
  Source_cell = Source_face = Adj_cell = Orient = -1;
};

Face::Face(const nodeindex ni0, const nodeindex ni1, const nodeindex ni2,
```

```
   cellindex source_cell, faceindex source_face,
   const vector<Node>& nt)
{
  // Sort the nodeindices and compute the orientation
  nodeindex tmp;
  Ni[0] = ni0;
  Ni[1] = ni1;
  Ni[2] = ni2;
  // starts with Orient 0 (rhs) assuming input with right hand out normal.
  Orient = 0;
  for (int i = 0; i < 3; i++)
    for (int j = i + 1; j < 3; j++)
      {
if (Ni[i] > Ni[j])
  {
    tmp = Ni[i];
    Ni[i] = Ni[j];
    Ni[j] = tmp;
    (Orient == 1) ? Orient = 0 : Orient = 1;
  };
      };
  Source_cell = source_cell;
  Source_face = source_face;
  Adj_cell = -1;

  // compute A, B, C, D - coefficients of the plane equation Ax+By+Cz=D
  // cross the two Positions to find normal (A, B, C) then plug in point
  // zero to find D.

  // Calculate the two Positions.
  Position pos0 = nt[ni0].pos();
  Position pos1 = nt[ni1].pos();
  Position pos2 = nt[ni2].pos();
  Position pos01 = pos1 - pos0;
  Position pos02 = pos2 - pos0;
  Normal = pos01.cross(pos02);
  Float scale_factor = Normal.magnitude();
  if (fabs(scale_factor) < EPSILON*EPSILON)
    {
      cerr << "Error: Trying to find plane from three colinear points.\n"
   << " Nodeindices " << ni0 << " " << ni1 << " " << ni2 << "\n";
      D = 0;
    }
  else
    {
      Normal.normalize();
      D = Normal.dot(nt[ni0].pos());
    };
};

Face::~Face()
{ };

Face& Face::operator=(const Face& f)
```

```
{
  if (this == &f) return *this;
  for (int i = 0; i < 3; i++) Ni[i] = f.Ni[i];
  Source_cell = f.Source_cell;
  Source_face = f.Source_face;
  Adj_cell = f.Adj_cell;
  Orient = f.Orient;
  Normal = f.Normal;
  D = f.D;
  return *this;
};

Face::Face(const Face& f)
{
  *this = f;
};

const cellindex Face::adj_cell(void) const
{
  return Adj_cell;
};

const cellindex Face::this_cell(void) const
{
  return Source_cell;
};

const nodeindex Face::ni(const int i) const
{
  return Ni[i];
};

const bool Face::exterior(void) const
{
  return (Adj_cell == -1);
};

const bool Face::check(Face& target)
{
  if (   Ni[0] == target.Ni[0]
      && Ni[1] == target.Ni[1]
      && Ni[2] == target.Ni[2])
    {
      Adj_cell = target.Source_cell;
      target.Adj_cell = Source_cell;
      return true;
    }
  else { return false; };
};

istream& operator>>(istream& s, Face& f)
{
  return s >> f.Ni[0]
           >> f.Ni[1]
```

```
           >> f.Ni[2]
           >> f.Source_cell
   >> f.Source_face
   >> f.Adj_cell
   >> f.Orient
   >> f.Normal
           >> f.D;
};

ostream& operator<<(ostream& s, const Face& f)
{
  return s << f.Ni[0] << " "
           << f.Ni[1] << " "
           << f.Ni[2] << "\n"
           << f.Source_cell << " "
   << f.Source_face << " "
   << f.Adj_cell << " "
   << f.Orient << "\n"
   << f.Normal << "\n"
           << f.D << "\n";
};

const int operator==(const Face&, const Face&)
{
  cerr << "Error, == not defined for Face.\n";
  return 0;
};

const int operator>(const Face&, const Face&)
{
  cerr << "Error, > not defined for Face.\n";
  return 0;
};

const int operator<(const Face&, const Face&)
{
  cerr << "Error, < not defined for Face.\n";
  return 0;
};

const int operator>=(const Face&, const Face&)
{
  cerr << "Error, >= not defined for Face.\n";
  return 0;
};

const int operator<=(const Face&, const Face&)
{
  cerr << "Error, <= not defined for Face.\n";
  return 0;
};

int hfunc(const Facep& f)
{
```

```
  int d1 = (f.ptr())->Ni[0]*2 + (f.ptr())->Ni[1]*3 + (f.ptr())->Ni[2]*5;
  return d1%N_of_Buckets;
};
```

A.16 Range.h

```
#ifndef __Range_h
#define __Range_h

#include <iostream.h>
#include "types.h"

/// A range of objects with a minimum and maximum.

/**
   The Range class maintains a range of values of some class type.
   The member add updates the range given the added object.  The
   member test determines whether the tested object is in range.  For
   non-scalar objects, these two members should be redefined.  In any
   case, all inequality and equality operators must be defined for the
   contained class.
*/

template<class T>
class Range
{
private:

friend istream& operator >>(istream& s, Range<T>& r);
friend ostream& operator <<(ostream& s, const Range<T>& r);

  T Min;               // Minimum element.
  T Max;               // Maximum element.
  int Touched;         // Has the range been explicitly set?

public:

  //  Default constructor.
  Range();

  ///  Explicit constructor takes initial min and max values.
  Range(T min, T max);

    //  Assignment operator.
  Range& operator=(const Range& r);

    //  Copy constructor
  Range(const Range& r);

  //  Destructor.
  ~Range();

  /// Returns minimum element.
```

```
  inline const T& min(void) const;

  /// Returns maximum element.
  inline const T& max(void) const;

  /// Changes and returns writeable minimum element.
  T& min(const T& newmin);

  /// Changes and returns writeable maximum element.
  T& max(const T& newmax);

  /// Updates the range based on a new sample.
  Range<T>& add(const T& item);

  /// Updates the range based on a new sample.
  Range<T>& operator+=(const T& elem);

  ///  Returns result of in range test on item.
  const bool test(const T& item) const;

  /// Trims an item to the range.
  /**
    If item is in range, truncate returns a copy of it.  If item
    is out of range, trunc returns a min or max object, whichever
    is closer to item.
    */
  const T truncate(const T& item) const;

  /// Returns the fractional value (0->1) of item within the range.
  const Float fraction(const T& item) const;

  /// Prints the range status to the standard error stream.
  void status(void) const;

};

template<class T>
istream& operator>>(istream& s, Range<T>& r);

template<class T>
ostream& operator <<(ostream& s, const Range<T>& r);

template<class T>
int operator==(Range<T>& r1, Range<T>& r2);

template<class T>
istream& operator>>(istream& s, Range<T>& r)
{
  T themin;
  T themax;
  bool touched;
  s >> themin >> themax >> touched;
  r.min(themin);
  r.max(themax);
```

```
  r.Touched = touched;
  return s;
};

template<class T>
ostream& operator <<(ostream& s, const Range<T>& r)
{
  return s << r.min() << "  " << r.max() << " " << r.Touched << "\n";
};

template<class T>
int operator==(Range<T>& r1, Range<T>& r2)
{
  return (r1.min() == r2.min()) && (r1.max() == r2.max());
};

template<class T>
inline const T& Range<T>::min(void) const
{
  return Min;
};

template<class T>
inline const T& Range<T>::max(void) const
{
  return Max;
};

#endif  /*  !__Range_h */
```

A.17 Range.cc

```
#ifndef __Range_cc
#define __Range_cc

#include <iostream.h>

#include "Range.h"

template<class T>
Range<T>::Range()
{
  Touched = 0;
};

template<class T>
Range<T>::Range(T min, T max)
{
  Min = min;
  Max = max;
  Touched = 1;
};
```

```
template<class T>
Range<T>& Range<T>::operator=(const Range& r)
{
  if (this == &r) return *this;
  Min = r.Min;
  Max = r.Max;
  Touched = r.Touched;
  return *this;
};

template<class T>
Range<T>::Range(const Range& r)
{
  *this = r;
};

template<class T>
Range<T>::~Range()
{ };

template<class T>
T& Range<T>::min(const T& newmin)
{
  Min = newmin;
  return Min;
};

template<class T>
T& Range<T>::max(const T& newmax)
{
  Max = newmax;
  return Max;
};

template<class T>
Range<T>& Range<T>::operator+=(const T& elem)
{
  add(elem);
  return *this;
};

template<class T>
void Range<T>::status(void) const
{
  cerr << " " << Min
       << "     " << Max
       << "\n";
};

Range<double>& Range<double>::add(const double& item)
{
  if (Touched)
    {
      if (item > Max)
```

```
{ Max = item; }
      else
{
  if (item < Min)
    Min = item;
};
    }
  else
    {
      Min = Max = item;
      Touched = 1;
    };
  return *this;
};

const bool Range<double>::test(const double& item) const
{
  return item >= Min && item <= Max;
};

const double Range<double>::truncate(const double& item) const
{
  if (item < Min)
    { return Min; }
  else if (item > Max)
    { return Max; }
  else return item;
};

const Float Range<double>::fraction(const double& item) const
{
  return (item - Min) / (Max - Min);
};

// float

Range<float>& Range<float>::add(const float& item)
{
  if (Touched)
    {
      if (item > Max)
{ Max = item; }
      else
{
  if (item < Min)
    Min = item;
};
    }
  else
    {
      Min = Max = item;
      Touched = 1;
    };
  return *this;
```

```
};

const bool Range<float>::test(const float& item) const
{
  return item >= Min && item <= Max;
};

const float Range<float>::truncate(const float& item) const
{
  if (item < Min)
    { return Min; }
  else if (item > Max)
    { return Max; }
  else return item;
};

const Float Range<float>::fraction(const float& item) const
{
  return (item - Min) / (Max - Min);
};

// Explicit instantiation used for Virtobj, so I don´t have to recompile
// Range in that library and suffer all of the function collisions.
// There should be a better way to do this, maybe someday...

template class Range<float>;

// Position

#include "Position.h"

Range<Position>& Range<Position>::add(const Position& item)
{
  if (Touched)
    {
      if (item.x() > Max.x())
{ Max = Position(item.x(), Max.y(), Max.z()); }
      else
if (item.x() < Min.x())
  { Min = Position(item.x(), Min.y(), Min.z()); };
      if (item.y() > Max.y())
{ Max = Position(Max.x(), item.y(), Max.z()); }
      else
if (item.y() < Min.y())
  { Min = Position(Min.x(), item.y(), Min.z()); };
      if (item.z() > Max.z())
{ Max = Position(Max.x(), Max.y(), item.z()); }
      else
if (item.z() < Min.z())
  { Min = Position(Min.x(), Min.y(), item.z()); };
    }
  else
    {
      Min = Max = item;
```

```
      Touched = 1;
    };
  return *this;
};

const bool Range<Position>::test(const Position& item) const
{
  return    (item.x() <= Max.x())
         && (item.y() <= Max.y())
 && (item.z() <= Max.z())
         && (item.x() >= Min.x())
         && (item.y() >= Min.y())
         && (item.z() >= Min.z());
};

const Position Range<Position>::truncate(const Position& item) const
{
  cerr << "Warning, calling Range<Position>:truncate(...)\n";
  return item;
};

const Float Range<Position>::fraction(const Position&) const
{
  cerr << "Warning, calling Range<Position>:fraction(...)\n";
  return 0.0;
};

// PosTimestep

#include "Position.h"

Range<PosTimestep>& Range<PosTimestep>::add(const PosTimestep& item)
{
  if (Touched)
    {
      int mints = Min.timestep();
      int maxts = Max.timestep();
      if (item.timestep() < Min.timestep())
{ mints = item.timestep(); }
      else if (item.timestep() > Max.timestep())
{ maxts = item.timestep(); };
      if (item.position().x() > Max.position().x())
{ Max = PosTimestep(Position(item.position().x(),
     Max.position().y(),
     Max.position().z()), maxts); }
      else
if (item.position().x() < Min.position().x())
  { Min = PosTimestep(Position(item.position().x(),
       Min.position().y(),
       Min.position().z()), mints); };
      if (item.position().y() > Max.position().y())
{ Max = PosTimestep(Position(Max.position().x(),
     item.position().y(),
     Max.position().z()), maxts); }
```

```
      else
if (item.position().y() < Min.position().y())
  { Min = PosTimestep(Position(Min.position().x(),
       item.position().y(),
       Min.position().z()), mints); };
      if (item.position().z() > Max.position().z())
{ Max = PosTimestep(Position(Max.position().x(),
     Max.position().y(),
     item.position().z()), maxts); }
      else
if (item.position().z() < Min.position().z())
  { Min = PosTimestep(Position(Min.position().x(),
       Min.position().y(),
       item.position().z()), mints); };
    }
  else
    {
      Min = Max = item;
      Touched = 1;
    };
  return *this;
};

const bool Range<PosTimestep>::test(const PosTimestep& item) const
{
  return    (item.position().x() <= Max.position().x())
         && (item.position().y() <= Max.position().y())
 && (item.position().z() <= Max.position().z())
 && (item.timestep() <= Max.timestep())
         && (item.position().x() >= Min.position().x())
         && (item.position().y() >= Min.position().y())
         && (item.position().z() >= Min.position().z())
         && (item.timestep() >= Min.timestep());
};

const PosTimestep Range<PosTimestep>::truncate(const PosTimestep& item) const
{
  cerr << "Warning, calling Range<PosTimestep>:truncate(...)\n";
  return item;
};

const Float Range<PosTimestep>::fraction(const PosTimestep&) const
{
  cerr << "Warning, calling Range<PosTimestep>:fraction(...)\n";
  return 0.0;
};

// shouldn´t need this explicit instn but g++ templates are still @#($*& up...
template class Range<PosTimestep>;

// Flowpoint

#include "Flowpoint.h"
```

```
Range<Flowpoint>& Range<Flowpoint>::add(const Flowpoint& item)
{
  if (Touched)
    {
      if (item.velocity().x() > Max.velocity().x())
{
  Position v(item.velocity().x(), Max.velocity().y(),
   Max.velocity().z());
  Max = Flowpoint(v, Max.pressure());
}
      else
if (item.velocity().x() < Min.velocity().x())
  {
    Position v(item.velocity().x(), Min.velocity().y(),
     Min.velocity().z());
    Min = Flowpoint(v, Min.pressure());
  };

      if (item.velocity().y() > Max.velocity().y())
{
  Position v(Max.velocity().x(), item.velocity().y(),
   Max.velocity().z());
  Max = Flowpoint(v, Max.pressure());
}
      else
if (item.velocity().y() < Min.velocity().y())
  {
    Position v(Min.velocity().x(), item.velocity().y(),
     Min.velocity().z());
    Min = Flowpoint(v, Min.pressure());
  };

      if (item.velocity().z() > Max.velocity().z())
{
  Position v(Max.velocity().x(), Max.velocity().y(),
   item.velocity().z());
  Max = Flowpoint(v, Max.pressure());
}
      else
if (item.velocity().z() < Min.velocity().z())
  {
    Position v(Min.velocity().x(), Min.velocity().y(),
     item.velocity().z());
    Min = Flowpoint(v, Min.pressure());
  };

      if (item.pressure() > Max.pressure())
{
  Position v(Max.velocity().x(), Max.velocity().y(),
   Max.velocity().z());
  Max = Flowpoint(v, item.pressure());
}
      else
if (item.pressure() < Min.pressure())
```

```
  {
    Position v(Min.velocity().x(), Min.velocity().y(),
     Min.velocity().z());
    Min = Flowpoint(v, item.pressure());
  };
    }
  else
    {
      Min = Max = item;
      Touched = 1;
    };
  return *this;
};

const bool Range<Flowpoint>::test(const Flowpoint& item) const
{
  return    (item.velocity().x() <= Max.velocity().x())
         && (item.velocity().y() <= Max.velocity().y())
 && (item.velocity().z() <= Max.velocity().z())
 && (item.pressure() <= Max.pressure())
         && (item.velocity().x() >= Min.velocity().x())
         && (item.velocity().y() >= Min.velocity().y())
         && (item.velocity().z() >= Min.velocity().z())
         && (item.pressure() >= Min.pressure());
};

const Flowpoint Range<Flowpoint>::truncate(const Flowpoint& item) const
{
  cerr << "Warning, calling Range<Flowpoint>:truncate(...)\n";
  return item;
};

const Float Range<Flowpoint>::fraction(const Flowpoint&) const
{
  cerr << "Warning, calling Range<Flowpoint>:fraction(...)\n";
  return 0.0;
};

// shouldn´t need this explicit instn but g++ templates are still @#($*& up...
template class Range<Flowpoint>;

#endif  /*  !__Range_cc */
```

A.18 Cell.h

```
#ifndef __Cell_h
#define __Cell_h

#include <iostream.h>
#include "types.h"

#include <vector.h>
```

```
#include "Range.h"

#include "Position.h"
#include "Node.h"
#include "Face.h"
#include "Linearfield.h"
#include "Flowpoint.h"

/// The fluid flow in a tetrahedron.
/**
  Represents flow within a tetrahedron over continuous space and time.
  Responsible for constructing fields for each timesteps, managing local
  adjacencies in the mesh; determining if positions are inside itself,
  and interpolating data at a given position and time.
*/

class Cell
{

private:

  friend istream& operator>>(istream& s, Cell& c);
  friend ostream& operator<<(ostream& s, const Cell& c);

  cellindex Cellindex;          // index number in some table
  nodeindex Nodelist[4];        // indices into a node table for 4 nodes
  Face Facelist[4];             // array of Cell´s four faces
  vector<Linearfield> Linflds;  // the linearfield of flowpoints in Cell
  Range<Position> Boundingbox;  // spatial bounding box around Cell

public:

    //  Default constructor, sets all nodeindices to zero.
  inline Cell();

    ///  Explicit constructor.
  Cell(const nodeindex node[4],
       const cellindex ci,
       const vector<Node>& nt);

    //  Assignment operator.
  Cell& operator=(const Cell& c);

    //  Copy constructor.
  inline
  Cell(const Cell& c);

    //  Destructor.
  inline
  ~Cell();

  void build_fields(const vector<Node>& nt);

    ///  Returns this cell´s index.
```

```
  inline const cellindex ci(void) const;

    ///  Returns ith nodeindex.
  inline const nodeindex nodeidx(const int i) const;

    ///  Returns ith adjacency.
  inline
  const cellindex adjacency(const int i) const;

    ///  Returns ith face.
  Face& face(const int i);

    ///  Changes and returns ref to ith nodeindex.
  nodeindex& nodeidx(const int i, const nodeindex ni);

    ///  Returns whether a position is inside the cell.
  const bool inside(const Position& pos) const;

    ///  Returns const reference to bounding box.
  inline
  const Range<Position>& boundingbox(void) const;

  const Linearfield& linfld(const int i) const;

  ///  Returns the value of a Flowpoint in the Cell´s field at a timestep
  inline
  const Flowpoint value(const Position& p, const int timestep) const;
};

  //  Read cell from input stream.
istream& operator>>(istream& s, Cell& c);

  //  Write cell to output stream.
ostream& operator<<(ostream& s, const Cell& c);

  // Equality operator == checks whether two cells are equal.
int operator==(const Cell& cellA, const Cell& cellB);


inline
Cell::Cell()
{ Nodelist[3] = Nodelist[2] = Nodelist[1] = Nodelist[0] = 0; };

inline
Cell::Cell(const Cell& c)
{ *this = c; };

inline
Cell::~Cell()
{ };

inline
const cellindex Cell::ci(void) const
{ return Cellindex; };
```

```
inline
const nodeindex Cell::nodeidx(const int i) const
{ return Nodelist[i]; };

inline
const Range<Position>& Cell::boundingbox(void) const
{ return Boundingbox; };

inline
const cellindex Cell::adjacency(const int i) const
{ return Facelist[i].adj_cell(); };

inline
nodeindex& Cell::nodeidx(const int i, const nodeindex ni)
{
  Nodelist[i] = ni;
  return Nodelist[i];
};

inline
const Flowpoint Cell::value(const Position& p, const int timestep) const
{
  return Linflds[timestep].value(p);
};

inline
const bool Cell::inside(const Position& pos) const
{
  return    Facelist[0].inside(pos)
         && Facelist[1].inside(pos)
         && Facelist[2].inside(pos)
         && Facelist[3].inside(pos);
};

#endif    /* !__Cell_h */
```

A.19 Cell.cc

```
#include "Cell.h"

Cell::Cell(const nodeindex node[4],
   const cellindex ci,
   const vector<Node>& nt)
{
  for (int i = 0; i < 4; i++)
    {
      Nodelist[i] = node[i];
      Boundingbox.add(nt[Nodelist[i]].pos());
    };
  Facelist[0] = Face(node[1], node[2], node[3], ci, 0, nt);
  Facelist[1] = Face(node[0], node[3], node[2], ci, 1, nt);
```

```
  Facelist[2] = Face(node[0], node[1], node[3], ci, 2, nt);
  Facelist[3] = Face(node[0], node[2], node[1], ci, 3, nt);
  Cellindex = ci;
};

Cell& Cell::operator=(const Cell& c)
{
  if (this == &c) return *this;
  Cellindex = c.Cellindex;
  for (int i = 0; i < 4; i++)
    {
      Nodelist[i] = c.Nodelist[i];
      Facelist[i] = c.Facelist[i];
    };
  Linflds = c.Linflds;
  Boundingbox = c.Boundingbox;
  return *this;
};

Face& Cell::face(const int i)
{
  return Facelist[i];
};

void Cell::build_fields(const vector<Node>& nt)
{
  int num_steps = nt[0].num_fp();
  Linflds.reserve(num_steps);
  Position posarr[4];
  for (int vx = 0; vx < 4; vx++)
    posarr[vx] = nt[Nodelist[vx]].pos();

  Flowpoint fparr[4];

  for (int ts = 0; ts < num_steps; ts++)
    {
      for (int vx = 0; vx < 4; vx++)
fparr[vx] = nt[Nodelist[vx]].fp(ts);
      Linflds.push_back(Linearfield(posarr, fparr));
    };
};

istream& operator>>(istream& s, Cell& c)
{
  c.Cell::~Cell();
  s >> c.Cellindex;
  for (int i = 0; i < 4; i++)
    s >> c.Nodelist[i];
  for (int i = 0; i < 4; i++)
    s >> c.Facelist[i];
  int size;
  s >> size;
  c.Linflds.reserve(size);
  for (int i = 0; i < size; i++)
```

```
    {
      Linearfield tmp;
      s >> tmp;
      c.Linflds.push_back(tmp);
    };
  s >> c.Boundingbox;
  return s;
};


ostream& operator<<(ostream& s, const Cell& c)
{
  s << c.Cellindex << "\n";
  for (int i = 0; i < 4; i++)
    s << c.Nodelist[i] << " ";
  for (int i = 0; i < 4; i++)
    s << c.Facelist[i] << " ";
  s << c.Linflds.size() << " ";
  for (unsigned int i = 0; i < c.Linflds.size(); i++)
    s << c.Linflds[i] << " ";
  s << c.Boundingbox << "\n";
  return s;
};

int operator==(const Cell& cellA,
       const Cell& cellB)
{
  return (cellA.nodeidx(0) == cellB.nodeidx(0)) &&
         (cellA.nodeidx(1) == cellB.nodeidx(1)) &&
         (cellA.nodeidx(2) == cellB.nodeidx(2)) &&
         (cellA.nodeidx(3) == cellB.nodeidx(3));
};

const Linearfield& Cell::linfld(const int i) const
{
  return Linflds[i];
};
```

A.20 Geom.h

```
#ifndef _Geom_h
#define _Geom_h

#include "Position.h"
#include "types.h"

/// A three dimensional line segment.
class Lineseg
{
protected:

  Position P1;
  Position P2;
```

```
public:

  Lineseg() {};

  /// Construct line segment from two endpoints.
  Lineseg(const Position& p1, const Position& p2)
    { P1 = p1; P2 = p2; };

  /// First endpoint.
  const Position& p1(void) const
    { return P1; };

  /// Second endpoint.
  const Position& p2(void) const
    { return P2; };

};

/// A triangle in three space.
class Triangle
{

protected:

  Position P1;
  Position P2;
  Position P3;

public:

  Triangle() {};

  /// Build a trinangle given three vertices.
  Triangle(const Position& p1, const Position& p2, const Position& p3);

  /// Does the line segment intersect the triangle face.
  const bool pierce(const Lineseg& lineseg);

};

#endif /* !_Geom_h */
```

A.21 Geom.cc

```
#include "Geom.h"

Triangle::Triangle(const Position& p1, const Position& p2, const Position& p3)
{
  P1 = p1;
  P2 = p2;
  P3 = p3;
};
```

```
const bool Triangle::pierce(const Lineseg& lineseg)
{
  // the equation for the line segment is pl(t) = pl1 + t*(pl2-pl1)
  // where t varies from 0->1
  // the equation for the triangle is pt(u,v) = pt1 + u(pt2-pt1) + v(pt3-pt1)
  // where u,v vary from 0->1
  //
  // setting pl(t) = pl(u,v)
  // yields an equation of the form A*t + B*u + C*v = D
  // where A,B,C,D are 3-vectors (columns).
  //
  // this can be written as a matrix equation:
  //  [   ] [t]   [ ]
  //  [ABC] [u] = [D]
  //  [   ] [v] = [ ]
  //
  // which can be solved as
  //  [t]   [   ]-1 [ ]
  //  [u] = [ABC]   [D]
  //  [v] = [   ]   [ ]
  //
  // unless the determinant = 0, in which case the line and plane are ||
  //
  // finally, if 0.0 <= t,u,v <= 1.0 the line segment pierces the triangle.
  //
  // if you were to need the actual pierce position, you could plug t
  // back into the line segment equation.

  // setup the p variables from the line and plane
  // the pl´s are from the lineseg; pt´s from the triangle
  const Float pl1x = lineseg.p1().x();
  const Float pl1y = lineseg.p1().y();
  const Float pl1z = lineseg.p1().z();
  const Float pl2x = lineseg.p2().x();
  const Float pl2y = lineseg.p2().y();
  const Float pl2z = lineseg.p2().z();

  // this is the maple code which was used to generate my C code
  /*
     with(linalg);
     pl1 := matrix([[pl1x],[pl1y],[pl1z]]);
     pl2 := matrix([[pl2x],[pl2y],[pl2z]]);
     pt1 := matrix([[pt1x],[pt1y],[pt1z]]);
     pt2 := matrix([[pt2x],[pt2y],[pt2z]]);
     pt3 := matrix([[pt3x],[pt3y],[pt3z]]);
     A := add(pl2,-pl1);
     B := add(pt1,-pt2);
     C := add(pt1,-pt3);
     D := add(pt1,-pl1);
     ABC := augment(A,B,C);
     ABCI := inverse(ABC);
     ANSWER := multiply(ABCI,D);
     readlib(C);
```

```
   C(ANSWER,optimized);
   quit;
   */

//calculate the pierce temporaries which are independent of the line

const Float t1 = P1.x()*P2.z();
const Float t3 = P2.x()*P1.z();
const Float t5 = P1.y()*P2.z();
const Float t8 = P1.x()*P2.y();
const Float t11 = P2.y()*P1.z();
const Float t39 = P2.x()*P1.y();
const Float t60 = P2.y()*P3.z();
const Float t65 = P1.y()*P3.z();
const Float t68 = P3.x()*P1.z();
const Float t71 = P3.x()*P1.y();
const Float t74 = P3.y()*P2.z();
const Float t77 = P1.x()*P3.y();
const Float t79 = P2.x()*P3.z();
const Float t81 = P1.x()*P3.z();
const Float t84 = P3.x()*P2.z();
const Float t108 = P2.x()*P3.y();
const Float t111 = P3.x()*P2.y();

// calculate the temporaries which are dep´t on the line

const Float t2 = pl2y*t1;
const Float t4 = pl1y*t3;
const Float t6 = pl2x*t5;
const Float t9 = pl1z*t8;
const Float t10 = pl1x*t5;
const Float t12 = pl1x*t11;
const Float t18 = pl2z*pl1x;
const Float t22 = pl2z*pl1y;
const Float t28 = pl2y*pl1z;
const Float t32 = pl2y*t3;
const Float t34 = pl2x*t11;
const Float t36 = -P1.y()*t18;
const Float t37 = pl1y*t1;
const Float t40 = pl2z*t39;
const Float t41 = pl2z*t8;
const Float t44 = pl1z*t39;
const Float t47 = pl2y*P1.z()*pl1x;
const Float t49 = -P1.x()*t28;
const Float t50 = P1.x()*t22;
const Float t52 = pl2x*P1.y()*pl1z;
const Float t55 = -pl2x*P1.z()*pl1y;
const Float t62 = -pl1x*t60;
const Float t63 = -t4;
const Float t64 = -t9;
const Float t66 = pl1x*t65;
const Float t67 = -t10;
const Float t69 = pl1y*t68;
const Float t73 = -pl1z*t71;
```

```
  const Float t78 = pl1z*t77;
  const Float t80 = pl1y*t79;
  const Float t83 = -pl1y*t81;
  const Float t87 = -pl2y*t68;
  const Float t92 = pl2y*t81;
  const Float t93 = pl1x*t74;
  const Float t94 = P3.y()*P1.z();
  const Float t96 = -pl1x*t94;
  const Float t97 = pl2x*t94;
  const Float t101 = -pl2x*t65;
  const Float t103 = -pl1y*t84;
  const Float t106 = -pl2z*t77;
  const Float t110 = pl2z*t71;
  const Float t115 = -pl1z*t108;
  const Float t116 = pl1z*t111;

  // solve the determinant first, if it´s close to zero, we have a line
  // parallel to our triangle, so we have no pierce.
  const Float determinant = (-pl2y*t79-pl2x*t74+pl2y*t84+t12+t6+t66+t67+t69
     +t64+t62+t63+t103+t44+t101+t37-t40+t41+t32-t34
     +t116+t115+t96+t97+pl2x*t60+t92+t93+t78+t80+t73+
     t83+t87+t106+t110+pl2z*t108-pl2z*t111-t2);
  if (fabs(determinant) < EPSILON*EPSILON) return false;

  // otherwise we can continue and find out where (if) we pierced.
  const Float inverse_det = 1/determinant;

  // t varies from 0 to 1 from P1 to P2 on the line segment
  const Float t = ( t62+t63+P3.x()*t5-P2.x()*t65-P3.y()*t1+P2.y()*t81+t64+t66
   +t67+t12+t69+t73-P3.x()*t11+P2.x()*t94+t78+t80+t83+t93
   +t96+t37+t103+t44+t115+t116
   )*inverse_det;

  if (t < -EPSILON*EPSILON || t > 1.0+EPSILON*EPSILON) return false;

  // u varies from 0 to 1 from P1 to P2 on the triangle
  const Float u = ( t66+t69+t73+t78+t83+t87+t92+t96+t97+t101+t36
   +t106+t110-pl2y*P3.z()*pl1x+t47+P3.x()*t28+t49+t50
   -P3.x()*t22+t52-pl2x*P3.y()*pl1z+t55+pl2x*P3.z()*pl1y
   +P3.y()*t18
   )*inverse_det;

  if (u < -EPSILON*EPSILON || u > 1.0+EPSILON*EPSILON) return false;

  // v varies from 0 to 1 from P1 to P3 on the triangle
  const Float v = -( t2+t4-t6+t9+t10-t12-pl2y*P2.z()*pl1x+P2.y()*t18
    +pl2x*P2.z()*pl1y-P2.x()*t22-pl2x*P2.y()*pl1z+P2.x()*t28
    -t32+t34+t36-t37+t40-t41-t44+t47+t49+t50
    +t52+t55
    )*inverse_det;

  return (v > -EPSILON*EPSILON && v < 1.0+EPSILON*EPSILON);

};
```

A.22 Spatialindex.h

```
#ifndef __Spatialindex_h
#define __Spatialindex_h

#include <iostream.h>
#include <vector.h>
#include <list.h>

#include "Range.h"
#include "Position.h"
#include "Cell.h"
#include "Geom.h"

/// Indexes cells by their spatial location.
/**
  Returns a list of cells near a position.  Constructs and interfaces a
  three dimensional index of cells.
*/

class Spatialindex
{
private:

  friend istream& operator >>(istream& s, Spatialindex& si);
  friend ostream& operator <<(ostream& s, const Spatialindex& si);

  vector<list<cellindex> > Index;
  int Xticks;
  int Yticks;
  int Zticks;
  Range<Position> Rp;

  int Max_cellrefs;
  int Total_cellrefs;
  int Buckets_occupied;

  list<cellindex> Null_table;

  const cellindex index_of(const Position& p) const;

  Float Minpx; // cached min x value of data
  Float Minpy; // cached min y value of data
  Float Minpz; // cached min z value of data
  Float Ranpx; // cached x width of data
  Float Ranpy; // cached y width of data
  Float Ranpz; // cached z width of data

public:

    // Default constructor.
  Spatialindex();

  /// Explicit constructor.
```

```
  Spatialindex(const Range<Position>& rp,
       const vector<Cell>& ct,
       const vector<Node>& nt,
       const int xticks, const int yticks, const int zticks);

    // Assignment operator.
  Spatialindex& operator=(const Spatialindex& si);

    // Copy constructor.
  Spatialindex(const Spatialindex& si);

    // Destructor.
  ~Spatialindex();

  /// Returns a vector of cellindices which are close to a position.
  const list<cellindex>& nearby_cells(const Position& p) const;

  /// Prints status to standard error stream.
  void status(void) const;

};

istream& operator>>(istream& s, Spatialindex& si);

ostream& operator<<(ostream& s, const Spatialindex& si);

inline
Spatialindex::Spatialindex()
: Index(), Xticks(0), Yticks(0), Zticks(0),
  Max_cellrefs(0), Total_cellrefs(0), Buckets_occupied(0), Null_table()
{ };

inline
Spatialindex::Spatialindex(const Spatialindex& si)
{
  *this = si;
};

inline
const list<cellindex>& Spatialindex::nearby_cells(const Position& p) const
{
  if (Rp.test(p))
    return Index[index_of(p)];
  else
    return Null_table;
  // should throw an exception here!!!
};

#endif   /* !__Spatialindex_h */
```

A.23 Spatialindex.cc

```
#include "Spatialindex.h"
```

```
const cellindex Spatialindex::index_of(const Position& p) const
{
  const int x = (int)( (p.x() - Minpx) / Ranpx * Xticks);
  const int y = (int)( (p.y() - Minpy) / Ranpy * Yticks);
  const int z = (int)( (p.z() - Minpz) / Ranpz * Zticks);
  return Xticks*(z*Yticks + y) + x;
};

Spatialindex::Spatialindex(const Range<Position>& rp,
   const vector<Cell>& ct,
   const vector<Node>& /* nt */,
   const int xticks,
   const int yticks,
   const int zticks)
: Index(xticks*yticks*zticks),
  Xticks(xticks), Yticks(yticks), Zticks(zticks),
  Rp(rp), Max_cellrefs(0), Total_cellrefs(0), Buckets_occupied(0),
  Null_table()
{
  Minpx = Rp.min().x();
  Minpy = Rp.min().y();
  Minpz = Rp.min().z();
  Ranpx = Rp.max().x() - Minpx;
  Ranpy = Rp.max().y() - Minpy;
  Ranpz = Rp.max().z() - Minpz;

  // the sizes of the "hash cubes"
  const Float size_x_tick = Ranpx / Xticks;
  const Float size_y_tick = Ranpy / Yticks;
  const Float size_z_tick = Ranpz / Zticks;

  // loop through each tetrahedral cell, find which hash cubes
  // intersect with it, and add the cell index to the table entry
  // for that cube
  for (cellindex ci = 0; (unsigned int)ci < ct.size(); ci++)
    {
      // the cell bounding box.
      Range<Position> cellbb = ct[ci].boundingbox();

      // these bounds are for the cell we are working with
      const Float xmin = cellbb.min().x();
      const Float ymin = cellbb.min().y();
      const Float zmin = cellbb.min().z();
      const Float xmax = cellbb.max().x();
      const Float ymax = cellbb.max().y();
      const Float zmax = cellbb.max().z();

      // these index bounds are the hashtable boxes corr. to the cell bb.
      const int i_start = (int)((xmin - Minpx) / size_x_tick);
      const int j_start = (int)((ymin - Minpy) / size_y_tick);
      const int k_start = (int)((zmin - Minpz) / size_z_tick);
      const int i_end = (int)((xmax - EPSILON - Minpx) / size_x_tick);
      const int j_end = (int)((ymax - EPSILON - Minpy) / size_y_tick);
```

```
      const int k_end = (int)((zmax - EPSILON - Minpz) / size_z_tick);

      // loop through all of the hash cubes which intersect with
      // the cell´s bounding box.
      for (int i = i_start; i <= i_end; i++)
for (int j = j_start; j <= j_end; j++)
  for (int k = k_start; k <= k_end; k++)
    {
      Index[Xticks*(k*Yticks+j)+i].push_back(ci);
    };
    };

  // compute statistics;
  Buckets_occupied = 0;
  Max_cellrefs = 0;
  Total_cellrefs = 0;

  for (int i = 0; i < Xticks*Yticks*Zticks; i++)
    {
      int cells_in_bucket = Index[i].size();
      if (cells_in_bucket > Max_cellrefs) Max_cellrefs = cells_in_bucket;
      if (cells_in_bucket > 0)
{
  Buckets_occupied++;
  Total_cellrefs += cells_in_bucket;
};
    };
};

Spatialindex& Spatialindex::operator=(const Spatialindex& si)
{
  if (this == &si) return *this;
  Index = si.Index;
  Xticks = si.Xticks;
  Yticks = si.Yticks;
  Zticks = si.Zticks;
  Rp = si.Rp;
  Minpx = si.Minpx;
  Minpy = si.Minpy;
  Minpz = si.Minpz;
  Ranpx = si.Ranpx;
  Ranpy = si.Ranpy;
  Ranpz = si.Ranpz;
  Max_cellrefs = si.Max_cellrefs;
  Total_cellrefs = si.Total_cellrefs;
  Buckets_occupied = si.Buckets_occupied;
  return *this;
};

void Spatialindex::status(void) const
{
  cerr << "  Xticks: " << Xticks
       << ", Yticks: " << Yticks
       << ", Zticks: " << Zticks << "\n"
```

```
       << " Position range: ";
  Rp.status();
  cerr << " Index size=" << Index.size() << ".\n";
  cerr << "Highest count in a cube             :" << Max_cellrefs << "\n";
  cerr << "Average count in all occupied cubes :"
       << ((Float)Total_cellrefs)/((Float)Buckets_occupied) << "\n";
  cerr << "Fraction of cubes occupied          :"
       << ((Float)Buckets_occupied)/((Float)(Xticks*Yticks*Zticks))
       << "\n" << flush;
};


istream& operator>>(istream& s, Spatialindex& si)
{
  int size;
  s >> size;
  si.Index = vector<list<cellindex> >(size);
  for (int i = 0; i < size; i++)
    {
      int listsize;
      s >> listsize;
      for (int j = 0; j < listsize; j++)
{
  cellindex tmp;
  s >> tmp;
  si.Index[i].push_back(tmp);
};
    };
  s >> si.Xticks;
  s >> si.Yticks;
  s >> si.Zticks;
  s >> si.Rp;
  si.Minpx = si.Rp.min().x();
  si.Minpy = si.Rp.min().y();
  si.Minpz = si.Rp.min().z();
  si.Ranpx = si.Rp.max().x() - si.Minpx;
  si.Ranpy = si.Rp.max().y() - si.Minpy;
  si.Ranpz = si.Rp.max().z() - si.Minpz;
  s >> si.Max_cellrefs;
  s >> si.Total_cellrefs;
  s >> si.Buckets_occupied;
  return s;
};

ostream& operator<<(ostream& s, const Spatialindex& si)
{
  s << si.Index.size() << "\n";
  for (unsigned int i = 0; i < si.Index.size(); i++)
    {
      s << si.Index[i].size() << "  ";
      list<cellindex>::const_iterator where;
      where = (si.Index[i]).begin();
      while (where != si.Index[i].end())
s << *where++ << " ";
```

```
      s << "\n";
    };
  s << si.Xticks << " ";
  s << si.Yticks << " ";
  s << si.Zticks << "\n";
  s << si.Rp << "\n";
  s << si.Max_cellrefs << " ";
  s << si.Total_cellrefs << " ";
  s << si.Buckets_occupied << "\n";
  return s;
};

Spatialindex::~Spatialindex()
{ };
```

A.24 Hashtable.h

```
#ifndef __Hashtable_h
#define __Hashtable_h

#include <iostream.h>
#include <vector.h>

template<class T>
class Hashtable
{
private:

  friend istream& operator >>(istream& s, Hashtable<T>& ht);
  friend ostream& operator <<(ostream& s, const Hashtable<T>& ht);

  vector<T>* Bucket_table;
  int Num_buckets;
  int (*Hashfunction)(const T& item);

public:

    // Default constructor.
  Hashtable();

    // Explicit constructor.
  Hashtable(int numbuckets,
    int (*hashfunction)(const T& item));

    // Assignment operator.
  Hashtable& operator=(const Hashtable& ht);

    // Copy constructor.
  Hashtable(const Hashtable& ht);

    // Destructor.
  ~Hashtable();
```

```
    // Adds an item to the table, using the hashfunction.
  const int add(const T& item);

    // Adds an item to a particular bucket, overriding the hashfunction.
  void force(const T& item, const int bucket_num);

  vector<T>& bucket(const int index);

  void status(void) const;

};

template<class T>
istream& operator >>(istream& s, Hashtable<T>& ht);

template<class T>
ostream& operator <<(ostream& s, const Hashtable<T>& ht);

#include "Hashtable.cc"

#endif  /* !__Hashtable_h */
```

A.25 Hashtable.cc

```
#ifndef __Hashtable_cc
#define __Hashtable_cc

#include "Hashtable.h"

template<class T>
Hashtable<T>::Hashtable()
{
  Bucket_table = 0;
  Num_buckets = 0;
  Hashfunction = 0;
};

template<class T>
Hashtable<T>::Hashtable(int numbuckets,
int (*hashfunction)(const T& item))
{
  Num_buckets = numbuckets;
  Hashfunction = hashfunction;
  Bucket_table = new vector<T> [Num_buckets];
};

template<class T>
Hashtable<T>& Hashtable<T>::operator=(const Hashtable<T>& ht)
{
  if (this == &ht) return *this;
  this->Hashtable::~Hashtable();
  Num_buckets = ht.Num_buckets;
  Hashfunction = ht.Hashfunction;
```

```
  Bucket_table = new vector<T> [Num_buckets];
  for (int i = 0; i < Num_buckets; i++)
    {
      Bucket_table[i] = ht.Bucket_table[i];
    };
  return *this;
};

template<class T>
Hashtable<T>::Hashtable(const Hashtable& ht)
{
  this->Bucket_table = 0;
  *this = ht;
};

template<class T>
Hashtable<T>::~Hashtable()
{
  delete [] Bucket_table;
  Bucket_table = 0;
};

template<class T>
const int Hashtable<T>::add(const T& item)
{
  int hf = Hashfunction(item);
  Bucket_table[hf].push_back(item);
  return hf;
};

template<class T>
void Hashtable<T>::force(const T& item, const int bucket_num)
{
  Bucket_table[bucket_num].push_back(item);
};

template<class T>
vector<T>& Hashtable<T>::bucket(const int index)
{
  return Bucket_table[index];
};

template<class T>
void Hashtable<T>::status(void) const
{
  cerr << " Hashtable: Bucket_table Head: " << Bucket_table;
  cerr << ", Num_buckets: " << Num_buckets
    << "\n";
};

template<class T>
istream& operator >>(istream& s, Hashtable<T>& ht)
{
  ht.Hashtable<T>::~Hashtable();
```

```
  s >> ht.Num_buckets;
  ht.Bucket_table = new vector<T>[ht.Num_buckets];
  for (int i = 0; i < ht.Num_buckets; i++)
    {
      vector<T> tmptable;
      int size;
      s >> size;
      tmptable.reserve(size);
      for (int i = 0; i < size; i++)
{
  T tmp;
  s >> tmp;
  tmptable.push_back(tmp);
};
      ht.Bucket_table[i] = tmptable;
    };
  return s;
};

template<class T>
ostream& operator <<(ostream& s, const Hashtable<T>& ht)
{
  s << ht.Num_buckets << "\n";
  for (int i = 0; i < ht.Num_buckets; i++)
    {
      int size = ht.Bucket_table[i].size();
      s << size << " ";
      for (int j = 0; j < size; j++)
s << ht.Bucket_table[i][j] << "\n";
    };
  return s;
};

#endif  /* !__Hashtable_cc */
```

A.26 mystl.h

```
#ifndef __mystl_h
#define __mystl_h

#include <vector.h>

template<class T>
istream& operator>>(istream& is, vector<T>& v)
{
  unsigned int numelem;
  is >> numelem;
  v.reserve(numelem);
  for (unsigned int i = 0; i < numelem; i++)
    {
      T tmp;
      is >> tmp;
      v.push_back(tmp);
```

```
    };
  return is;
};

template<class T>
ostream& operator<<(ostream& os, const vector<T>& v)
{
  os << v.size() << "\n";
  for (unsigned int i = 0; i < v.size(); i++)
    {
      os << v[i] << "\n";
    };
  return os;
};

#endif /* !__mystl_h */
```

A.27 Flowfield.h

```
#ifndef __Flowfield_h
#define __Flowfield_h

#include <iostream.h>
#include <ctype.h>
#include <fstream.h>
#include <math.h>

#include <algo.h>
#include <vector.h>

#include "Range.h"
#include "Hashtable.h"
#include "Node.h"
#include "types.h"
#include "mystl.h"
#include "Spatialindex.h"
#include "Cell.h"
#include "Flowpoint.h"
#include "Position.h"

#ifdef USES_THREADS
extern "C"
{
#include <thread.h>
#include <unistd.h>
};
// class OutOfThreads { }; // exception
#endif /* USES_THREADS */

/// Fluid flow model continuous in space and time.
/**
Represents fluid flow in continuous space and time over a three
dimensional region.  Construct and organizes cells and nodes in
```

```
tables.  Reads in and digests Nekton data files.  Performs global and
local spatial/temporal interpolation of flow data.
*/

class Flowfield

{

private:

  friend istream& operator>>(istream& s, Flowfield& f);
  friend ostream& operator<<(ostream& s, const Flowfield& f);
  friend Flowfield operator-(const Flowfield& f1, const Flowfield& f2);
  Range<PosTimestep> PosTimesteprange;
  Range<Flowpoint> Flowpointrange;
  vector<Range<Flowpoint> > StepwiseFlowpointrange;
  vector<Node> Nodes;           // all of the nodes
  vector<Cell> Cells;           // all of the cells, which index to the nodes
  vector<Face> Exteriors;       // the exterior faces of the flowfield
  vector<Float> Times;          // the real times of the timesteps
  Range<Float> Speedrange;
  vector<Range<Float> > StepwiseSpeedrange;
  Range<Float> Absstrainraterange;
  vector<Range<Float> > StepwiseAbsstrainraterange;
  Range<Float> Divergencerange;
  vector<Range<Float> > StepwiseDivergencerange;
  Range<Float> Surfstrainraterange;
  vector<Range<Float> > StepwiseSurfstrainraterange;
  Spatialindex Spindex;

  int Cur_time_step;
  int Cindex;           // keeps track of the next cellindex

    //  new_time should be called once at the beginning of every time step,
    //  and be passed the value of the time.
  void New_time(const Float& time);

    //  add_cell is the general member for adding cells at any time.
    //  It takes an array of four nodes which should have their first
    //  Flowpoint equal to the Flowpoint at the current time.
  void Add_cell(Node nodes[4]);

  void Make_adjacencies(void);
  void Build_exteriors(void);
  void Build_cell_fields(void);

    //  lock should be called after all cells at all times have been
    //  added.  It makes sure all tables of Flowfield are locked, and
    //  determines adjacencies and exteriors.
  void Lock(void);

  // Used when we know exactly which cell we are in.
  const Flowpoint Interpolant(const Position& pos,
      const Float& time,
```

```
      cellindex& ci) const;

  const Flowpoint Interpolant(const Position& pos,
      const int timestep,
      cellindex& ci) const;

  void Add_333_block(const Node block[27]);

public:

  // Default constructor;
  Flowfield();

  //  Assignment operator.
  Flowfield& operator=(const Flowfield& f);

  //  Copy constructor.
  Flowfield(const Flowfield& f);

  //  Destructor.
  ~Flowfield();

  /// Prints the Flowfield status to standard error stream.
  void status(void) const;

  /// Returns ith node.
  const Node& node(const int i) const;

  /// Returns how many nodes exist.
  const int num_node(void) const;

  /// Returns ith cell.
  const Cell& cell(const int i) const;

  /// Returns how many cells exist.
  const int num_cell(void) const;

  /// Returns ith exterior face.
  const Face& exterior(const int i) const;

  /// Returns how many exterior faces exist.
  const int num_exterior(void) const;

  /// Returns timestamp value of ith timestep.
  const Float& time(const int i) const;

  /// Returns number of timesteps.
  const int num_time(void) const;

  const Range<PosTimestep>& postimesteprange(void) const;

  const Range<Flowpoint>& flowpointrange(void) const;

  /// Returns range of speed values.
```

```
  const Range<Float>& speedrange(void) const;

  /// Returns range of absolute strain rate values.
  const Range<Float>& absstrainraterange(void) const;

  /// Returns range of divergence values.
  const Range<Float>& divergencerange(void) const;

  /// Returns range of surface strain rate values
  const Range<Float>& surfstrainraterange(void) const;

  // Interpolaters.
  /// Returns the interpolated flow value at a position and timestep.
  const Flowpoint eval(const PosTimestep& pt) const;

#ifdef USES_THREADS
  // the threading version of the timestep based interpolator
  void eval(const int size,
    const Position pos[],
    const int timestep,
    Flowpoint fp[])
    const
;

#endif /* USES_THREADS */

  /// Builds flowfield from Nekton field file with given sampling.
  istream& read_hex_mesh(istream& fld_file, const bool sparse);

};

  ///  Build Flowfield from .ff format input stream.
istream& operator>>(istream& s, Flowfield& f);

  ///  Write Flowfield to .ff formatted output stream.
ostream& operator<<(ostream& s, const Flowfield& f);

//  Construct flowfield as difference of two flowfield
Flowfield operator-(const Flowfield& f1, const Flowfield& f2);

inline
Flowfield::~Flowfield()
{ };

inline
Flowfield::Flowfield(const Flowfield& f)
{ *this = f; };

inline
const Node& Flowfield::node(const int i) const
{ return Nodes[i]; };

inline
const int Flowfield::num_node(void) const
```

```
{ return Nodes.size(); };

inline
const Cell& Flowfield::cell(const int i) const
{ return Cells[i]; };

inline
const int Flowfield::num_cell(void) const
{ return Cells.size(); };

inline
const Face& Flowfield::exterior(const int i) const
{ return Exteriors[i]; };

inline
const int Flowfield::num_exterior(void) const
{ return Exteriors.size(); };

inline
const Float& Flowfield::time(const int i) const
{ return Times[i]; };

inline
const int Flowfield::num_time(void) const
{ return Times.size(); };

inline
const Range<PosTimestep>& Flowfield::postimesteprange(void) const
{ return PosTimesteprange; };

inline
const Range<Flowpoint>& Flowfield::flowpointrange(void) const
{ return Flowpointrange; };

inline
const Range<Float>& Flowfield::speedrange(void) const
{ return Speedrange; };

inline
const Range<Float>& Flowfield::absstrainraterange(void) const
{ return Absstrainraterange; };

inline
const Range<Float>& Flowfield::divergencerange(void) const
{ return Divergencerange; };

inline
const Range<Float>& Flowfield::surfstrainraterange(void) const
{ return Surfstrainraterange; };

#endif  /* !__Flowfield_h */
```

A.28 Flowfield.cc

```
#include "Flowfield.h"

void
Flowfield::New_time(const Float& time)
{
  // Add the time and update the current time step.
  Times.push_back(time);
  StepwiseFlowpointrange.push_back(Range<Flowpoint>());
  StepwiseSpeedrange.push_back(Range<Float>());
  StepwiseAbsstrainraterange.push_back(Range<Float>());
  StepwiseDivergencerange.push_back(Range<Float>());
  StepwiseSurfstrainraterange.push_back(Range<Float>());
  Cur_time_step++;
};

void
Flowfield::Add_cell(Node nodes[4])
{
  nodeindex ni[4];

  for (int i = 0; i < 4; i++)
    {
      // Add the node and get the index back.
      vector<Node>::iterator where;
      where = find(Nodes.begin(), Nodes.end() ,nodes[i]);
      if (where == Nodes.end())
{
  Nodes.push_back(nodes[i]);
  ni[i] = Nodes.size() - 1;
}
      else
{
  ni[i] = where - Nodes.begin();
};

      //  If we haven´t added this Flowpoint yet, then do it.
      if (Cur_time_step != Nodes[ni[i]].num_fp())
{
  Nodes[ni[i]].add_fp(nodes[i].fp(0));
  Flowpointrange.add(nodes[i].fp(0));
};
    };

  //  If we are on the first timestep, add the cell to
  //  the cells and update the global range.
  if (Cur_time_step == 1)
    {
      for (int idx = 0; idx < 4; idx++)
{
  PosTimestep tmppt(Nodes[ni[idx]].pos(), Cur_time_step);
  PosTimesteprange.add(tmppt);
};
```

```
      Cells.push_back(Cell(ni, Cindex++, Nodes));
    };
};

void
Flowfield::Make_adjacencies(void)
{
  //  Hashtable for finding face adjacencies
  Hashtable<Facep> Hface_table(N_of_Buckets, hfunc);

  //  Adding all Face pointers to Hashtable
  for (cellindex ci = 0; (unsigned int)ci < Cells.size(); ci++)
    for (faceindex fi = 0; fi < 4; fi++)
      Hface_table.add(Facep(Cells[ci].face(fi)));

  // Checking hashtable to find adjacencies in each bucket
  for (int b = 0; b < N_of_Buckets; b++)
    for (unsigned int s = 0; s < Hface_table.bucket(b).size(); s++)
      for (unsigned int t = s + 1; t < Hface_table.bucket(b).size(); t++)
if (
    (Hface_table.bucket(b)[s].ptr())
    ->check(*(Hface_table.bucket(b)[t].ptr()))
    )
  break;
};

void
Flowfield::Build_exteriors(void)
{
  for (unsigned int ci = 0; ci < Cells.size(); ci++)
    for (faceindex fi = 0; fi < 4; fi++)
      if (Cells[ci].face(fi).exterior())
{
  Exteriors.push_back(Cells[ci].face(fi));
};

  // compute surface strain rate range.
  for (int slice = 0; slice < num_time(); slice++)
    {
      for (int facenum = 0; facenum < num_exterior(); facenum++)
{
  for (int vertnum = 0; vertnum < 3; vertnum++)
    {
      Flowpoint fpt = node(exterior(facenum).ni(vertnum)).fp(slice);
      Linearfield lf = Cells[(exterior(facenum).this_cell())]
                     .linfld(slice);
      StrainRate sr = lf.strainrate();
      Flowpoint realfp(fpt.velocity(), fpt.pressure());
      realfp.setstrainrate(sr);
      Position norm = exterior(facenum).normal();
      Position surfstrainrate = realfp.surfstrainrate(norm);
      Float val = surfstrainrate.magnitude();
      Surfstrainraterange.add(val);
      StepwiseSurfstrainraterange[slice].add(val);
```

```
    };
};
    };
};

void
Flowfield::Build_cell_fields(void)
{
  for (unsigned int c = 0; c < Cells.size(); c++) {
    Cells[c].build_fields(Nodes);
    for (int t = 0; t < num_time(); t++) {
      Absstrainraterange.add(Cells[c].linfld(t).strainrate().absstrainrate());
      StepwiseAbsstrainraterange[t].add(Cells[c].linfld(t)
.strainrate().absstrainrate());
      Divergencerange.add(Cells[c].linfld(t).strainrate().divergence());
      StepwiseDivergencerange[t].add(Cells[c].linfld(t)
     .strainrate().divergence());
    };
  };
  for (unsigned int n = 0; n < Nodes.size(); n++) {
    for (int t = 0; t < num_time(); t++) {
      Speedrange.add(node(n).fp(t).speed());
      StepwiseSpeedrange[t].add(node(n).fp(t).speed());
      Flowpointrange.add(node(n).fp(t));
      StepwiseFlowpointrange[t].add(node(n).fp(t));
    };
  };
};

void
Flowfield::Lock(void)
{
  cerr << "Making Adjacencies.\n";
  Make_adjacencies();

  cerr << "Building Cell Fields.\n";
  Build_cell_fields();

  cerr << "Building Exteriors.\n";
  Build_exteriors();

  cerr << "Constructing Spatial index.\n";
  const int XTICKS = 25;
  const int YTICKS = 25;
  const int ZTICKS = 25;
  Range<Position> posdomain(PosTimesteprange.min().position(),
    PosTimesteprange.max().position());
  Spindex = Spatialindex(posdomain, Cells, Nodes,
 XTICKS, YTICKS, ZTICKS);
};

const Flowpoint
Flowfield::Interpolant(const Position& pos,
       const Float& time,
```

```
       cellindex& ci)
     const
{
  if (   time < Times[0]
      || time > Times[Times.size() - 1])
      {
cerr << "Error in Flowfield::Interpolant:\n"
     << " time = " << time << " out of range\n";
return Flowpoint(Position(), 0.0);
// should throw an exception here!!!
// note it would be nice to do limited extrapolation at this point.
      }
  int num_segments = Times.size() - 1;
  Float timerange =  Times[num_segments] - Times[0];
  Float frac_time_index = (time - Times[0])
                          / timerange * num_segments;
  int time_floor = (int)floor(frac_time_index);
  int time_ceil = (int)ceil(frac_time_index);
  Float weight =   (time - Times[time_floor])
                 / (Times[time_ceil] - Times[time_floor]);
  return (1 - weight)*(Cells[ci].value(pos, time_floor))
    + weight*(Cells[ci].value(pos, time_ceil));
};

const Flowpoint
Flowfield::Interpolant(const Position& pos,
       const int timestep,
       cellindex& ci)
     const
{
  if (   timestep < 0
      || (unsigned int)timestep >= Times.size()
      )
      {
cerr << "Error in Flowfield::Interpolant:\n"
     << " time = " << timestep << " out of range\n";
return Flowpoint();
// should throw exception!!!
      };
  return (Cells[ci].value(pos, timestep));
};

#ifdef USES_THREADS

struct
workticket{
  const Flowfield* ffp;   // pointer to the calling flowfield
  const Position* pos;    // start of my position array (input)
  int timestep;
  Flowpoint* fp;    // start of my flowpoint array (output)
  int howmany;      // how far do I work from the start?
  mutex_t* donelock;
  cond_t* done;
  int* activejobs;
```

```
};

void*
worker(void* wtv)
{
  const workticket& wt = *((workticket*) wtv);
  for (int i = 0; i < wt.howmany; i++)
    {
      wt.fp[i] = wt.ffp->eval(PosTimestep(wt.pos[i], wt.timestep));
    };
  mutex_lock(wt.donelock);
  (*(wt.activejobs))--;
  cond_signal(wt.done); // notify I´m done
  mutex_unlock(wt.donelock);
  return 0;
};

void
Flowfield::eval(const int size, const Position pos[],
const int timestep, Flowpoint fp[])
     const
//     throw (OutOfThreads)
{
  mutex_t donelock;
  mutex_init(&donelock, USYNC_THREAD, 0);
  cond_t done;
  cond_init(&done, USYNC_THREAD, 0);
  mutex_lock(&donelock);  // lock this up so no one can beat me to the exit.
  int activejobs = 0;
  const int numthreads = 5;
  int chunksize = (size/numthreads) + 1;
  int numchunks = (size/chunksize) + ((size%chunksize) > 0);
  workticket* wts[numchunks];
  int threadnum = 0;
  int worktodo = size;
  thr_setconcurrency(numthreads);
  while (worktodo > 0)
    {
      activejobs++;
      int thischunk = ((worktodo >= chunksize) ? chunksize : worktodo);
      wts[threadnum] = new workticket;
      wts[threadnum]->ffp = this;
      wts[threadnum]->pos = pos + (threadnum*chunksize);
      wts[threadnum]->timestep = timestep;
      wts[threadnum]->fp = fp + (threadnum*chunksize);
      wts[threadnum]->howmany = thischunk;
      wts[threadnum]->donelock = &donelock;
      wts[threadnum]->done = &done;
      wts[threadnum]->activejobs = &activejobs;
      //      int res =
      (void) thr_create(NULL, 0, worker, (void *)(wts[threadnum]),
THR_DETACHED, NULL);
      //      if (res) throw(OutOfThreads);
      //      if (res) cerr << "Error spawning thread." << flush;
```

```
      // should throw exception!!!
      worktodo -= thischunk;
      threadnum++;
    };

  while (activejobs)  // count down the jobs as our workers finish
    {
      cond_wait(&done, &donelock);
    };
  mutex_unlock(&donelock);
  for (int i = 0; i < numchunks; i++)
    {
      delete wts[i];
    };
  cond_destroy(&done);
  mutex_destroy(&donelock);
};

#endif /* USES_THREADS */

void Flowfield::Add_333_block(const Node block[27])
{
  //  Map for creating 48 tetrahedrons from a 3x3x3 hexahedral block.
  //  Each tetrahedron has one common, central point (13) which is
  //  excluded.  Points in block are sequentially numbered first
  //  by X, then Y, then Z - starting from zero.
  //  Faces are numbered in the comments by the node number at the center
  //  of the face.
  int tetmap[48][3] =
    {
      // face 4 (draw the picture)
      {4,0,3},    {4,3,6},    {4,6,7},    {4,7,8},
      {4,8,5},    {4,5,2},    {4,2,1},    {4,1,0},
      // face 10
      {10,19,18}, {10,20,19}, {10,11,20}, {10,2,11},
      {10,1,2},   {10,0,1},   {10,9,0},   {10,18,9},
      // face 12
      {12,21,24}, {12,18,21}, {12,9,18},  {12,0,9},
      {12,3,0},   {12,6,3},   {12,15,6},  {12,24,15},
      // face 14
      {14,23,20}, {14,26,23}, {14,17,26}, {14,8,17},
      {14,5,8},   {14,2,5},   {14,11,2},  {14,20,11},
      // face 16
      {16,24,25}, {16,25,26}, {16,26,17}, {16,17,8},
      {16,8,7},   {16,7,6},   {16,6,15},  {16,15,24},
      // face 22
      {22,21,18}, {22,24,21}, {22,25,24}, {22,26,25},
      {22,23,26}, {22,20,23}, {22,19,20}, {22,18,19}
    };

  Node tetnodes[4];
  tetnodes[3] = block[13];
  for (int i = 0; i < 48; i++)
    {
```

```
      for (int j = 0; j < 3; j++)
tetnodes[j] = block[tetmap[i][j]];
      Add_cell(tetnodes);
    };
};


Flowfield::Flowfield()
{
  Cur_time_step = 0;
  Cindex = 0;
};

Flowfield& Flowfield::operator=(const Flowfield& f)
{
  if (this == &f) return *this;
  PosTimesteprange = f.PosTimesteprange;
  Flowpointrange = f.Flowpointrange;
  StepwiseFlowpointrange = f.StepwiseFlowpointrange;
  Nodes = f.Nodes;
  Cells = f.Cells;
  Exteriors = f.Exteriors;
  Times = f.Times;
  Speedrange = f.Speedrange;
  StepwiseSpeedrange = f.StepwiseSpeedrange;
  Absstrainraterange = f.Absstrainraterange;
  StepwiseAbsstrainraterange = f.StepwiseAbsstrainraterange;
  Divergencerange = f.Divergencerange;
  StepwiseDivergencerange = f.StepwiseDivergencerange;
  Surfstrainraterange = f.Surfstrainraterange;
  StepwiseSurfstrainraterange = f.StepwiseSurfstrainraterange;
  Spindex = f.Spindex;
  Cur_time_step = f.Cur_time_step;
  Cindex = f.Cindex;
  return *this;
};

const Flowpoint Flowfield::eval(const PosTimestep& pt) const
{
  const list<cellindex>& candidates = Spindex.nearby_cells(pt.position());
  list<cellindex>::const_iterator where = candidates.begin();
  while (where != candidates.end())
    {
      cellindex check_cell = *where++;
      if (Cells[check_cell].inside(pt.position()))
{
  return Interpolant(pt.position(), pt.timestep(), check_cell);
};
    };
  // should throw exception, but they don´t work right!!!
  return Flowpoint();
};

void Flowfield::status(void) const
```

```
{
  cerr << "Nodes: " << Nodes.size() << "\n";
  cerr << "Cells: " << Cells.size() << "\n";
  cerr << "Exterior faces: " << Exteriors.size() << "\n";
  cerr << "Timesteps: " << Times.size() << "\n";
  cerr << "PosTimesteprange: ";
  PosTimesteprange.status();
  cerr << "Flowpointrange: ";
  Flowpointrange.status();
  cerr << "StepwiseFlowpointrange:\n";
  for (unsigned int i = 0; i < StepwiseFlowpointrange.size(); i++)
    {
      cerr << i << " : ";
      StepwiseFlowpointrange[i].status();
    };

  cerr << "Speedrange: ";
  Speedrange.status();
  cerr << "StepwiseSpeedrange:\n";
  for (unsigned int i = 0; i < StepwiseSpeedrange.size(); i++)
    {
      cerr << i << " : ";
      StepwiseSpeedrange[i].status();
    };

  cerr << "Absstrainraterange: ";
  Absstrainraterange.status();
  cerr << "StepwiseAbsstrainraterange:\n";
  for (unsigned int i = 0; i < StepwiseAbsstrainraterange.size(); i++)
    {
      cerr << i << " : ";
      StepwiseAbsstrainraterange[i].status();
    };
  cerr << "Divergencerange: ";
  Divergencerange.status();
  cerr << "StepwiseDivergencerange:\n";
  for (unsigned int i = 0; i < StepwiseDivergencerange.size(); i++)
    {
      cerr << i << " : ";
      StepwiseDivergencerange[i].status();
    };

  cerr << "Surfstrainraterange: ";
  Surfstrainraterange.status();
  cerr << "StepwiseSurfstrainraterange:\n";
  for (unsigned int i = 0; i < StepwiseSurfstrainraterange.size(); i++)
    {
      cerr << i << " : ";
      StepwiseSurfstrainraterange[i].status();
    };

  cerr << "Spindex: ";
  Spindex.status();
};
```

```
istream& operator>>(istream& s, Flowfield& f)
{
  cerr << " Reading nodes. ";
  s >> f.Nodes;
  cerr << " Reading cells. ";
  s >> f.Cells;
  s >> f.Exteriors;
  s >> f.Times;
  s >> f.Cur_time_step;
  s >> f.Cindex;
  s >> f.PosTimesteprange;
  s >> f.Flowpointrange;
  s >> f.StepwiseFlowpointrange;
  s >> f.Speedrange;
  s >> f.StepwiseSpeedrange;
  s >> f.Absstrainraterange;
  s >> f.StepwiseAbsstrainraterange;
  s >> f.Divergencerange;
  s >> f.StepwiseDivergencerange;
  s >> f.Surfstrainraterange;
  s >> f.StepwiseSurfstrainraterange;
  cerr << " Reading spatialindex. ";
  s >> f.Spindex;
  return s;
};

ostream& operator<<(ostream& s, const Flowfield& f)
{
  s << f.Nodes;
  s << f.Cells;
  s << f.Exteriors;
  s << f.Times;
  s << f.Cur_time_step << "\n";
  s << f.Cindex << "\n";
  s << f.PosTimesteprange << "\n";
  s << f.Flowpointrange << "\n";
  s << f.StepwiseFlowpointrange << "\n";
  s << f.Speedrange << "\n";
  s << f.StepwiseSpeedrange << "\n";
  s << f.Absstrainraterange << "\n";
  s << f.StepwiseAbsstrainraterange << "\n";
  s << f.Divergencerange << "\n";
  s << f.StepwiseDivergencerange << "\n";
  s << f.Surfstrainraterange << "\n";
  s << f.StepwiseSurfstrainraterange << "\n";
  s << f.Spindex << "\n";
  return s;
};

istream& Flowfield::read_hex_mesh(istream& fld_file, const bool sparse)
{
  int isize, jsize, ksize;   //  The dimensions of a block
  int numb_elem;             //  Number of blocks in timestep
```

```
  Float time;               //  The current time stamp

  cerr << "Beginning to read Nekton file.\n";

  //  Keep looping through timesteps until eof.
  while (!fld_file.eof())
  {
    //  Read the important part of the header
    fld_file >> numb_elem
             >> isize
     >> jsize
     >> ksize
     >> time;

    cerr << "Reading time " << time << "\n" << "Counting elements";

    //  Read and discard unimportant part of header.
    char cc;
    while (fld_file.get(cc) && cc != ´\n´);

    //  Read and discard the zeros in the header
    Float dummy;
    for (int ii = 0; ii < numb_elem; ii++)
      {
fld_file >> dummy;
      };

    New_time(time);    //  Initiate the new timestep.

    Node full_block[125];    // the full 5x5x5 Node hexahedral block

    //  Loop through all of the blocks.
    for (int blockcount = 0; blockcount < numb_elem; blockcount++)
      {
cerr << ".";

//  Loop through all 125 nodes in one block
for (int nodenum = 0; nodenum < 125; nodenum++)
  {
    const int SCALE = 1000;
    Float x;  Float y;  Float z;
    fld_file >> x >> y >> z;
    Position pos(x,y,z);
    pos = SCALE * pos;
    Node nod(pos);
    Float u;  Float v;  Float w;  Float p;
    fld_file >> u >> v >> w >> p;
    Flowpoint flowp(Position(u, v, w), p);
    nod.add_fp(flowp);
    full_block[nodenum] = nod;
  };
// at this point we have full block.

if (sparse)
```

```
  { // extract sparse block from full block
    Node sparse_block[27];   // the sparse 3x3x3 Node contained block
    int n = 0;
    int m = 0;
    for (int k = 0; k < 5; k = k + 2)
      for (int j = 0; j < 5; j = j + 2)
for (int i = 0; i < 5; i = i + 2)
  {
    m = k*25 + j*5 + i;
    sparse_block[n++] = full_block[m];
  };

    // build and add cells.
    Add_333_block(sparse_block);
  }
else
  { // divide full block into eight small blocks
    for (int k = 0; k < 4; k += 2)
      for (int j = 0; j < 4; j += 2)
for (int i = 0; i < 4; i += 2)
  {
    int n = 0;
    int m = 0;
    Node small_block[27];
    for (int kk = k; kk < k + 3; kk++)
      for (int jj = j; jj < j + 3; jj++)
for (int ii = i; ii < i + 3; ii++)
  {
    m = kk*25 + jj*5 + ii;
    small_block[n++] = full_block[m];
  };
    Add_333_block(small_block);
  };
  };
      };     // end loop blockcount

    //  Eat whitespaces.
    char ws;
    while (fld_file.get(ws))
      {
if (!isspace(ws))
  {
    fld_file.putback(ws);
    break;
  };
      };
    cerr << "\n";
  };

  cerr << "\nLocking Flowfield.\n";

  Lock();

  return fld_file;
```

```
};

Flowfield operator-(const Flowfield& f1, const Flowfield& f2)
{
  Flowfield f;
  assert(f1.num_cell()==f2.num_cell());
  for (int t = 0; t < f1.num_time(); t++) {
    f.New_time(f1.time(t));
    for (int c = 0; c < f1.num_cell(); c++) {
      Node nodes[4];
      for (int n = 0; n < 4; n++) {
const Node& f1node = f1.node(f1.cell(c).nodeidx(n));
const Node& f2node = f2.node(f2.cell(c).nodeidx(n));
nodes[n] = Node(f1node.pos());
nodes[n].add_fp(f1node.fp(t) - f2node.fp(t));
      };
      f.Add_cell(nodes);
    };
  };
  f.Lock();
  return f;
};
```

Appendix B

VirtualObject library source

B.1 utils.h

```
#ifndef __utils_h
#define __utils_h

#include "Position.h"
#include "Quaternion.h"
#include "./wt.h"

inline
void posconvert(const Position& ffp, WTp3& wtp)
{
  wtp[0] = ffp.x();
  wtp[1] = ffp.y();
  wtp[2] = ffp.z();
};

inline
void posconvert(const WTp3& wtp, Position& ffp)
{
  ffp = Position(wtp[0], wtp[1], wtp[2]);
};

inline
void oriconvert(const Quaternion& ffq, WTq& wtq)
{
  wtq[0] = ffq.axis().x();
  wtq[1] = ffq.axis().y();
  wtq[2] = ffq.axis().z();
  wtq[3] = ffq.w();
};

inline
void oriconvert(const WTq& wtq, Quaternion& ffq)
{
  ffq = Quaternion(wtq[3], Position(wtq[0], wtq[1], wtq[2]));
};

#endif /* !__utils_h */
```

B.2 Colormap.h

```
#ifndef __Colormap_h
#define __Colormap_h

#include <iostream.h>
#include "../Flowfield/types.h"
#include "../Flowfield/Range.h"

/// 24 bit RGB color

class Color
{
private:

  friend istream& operator>>(istream& s, Color& c);
  friend ostream& operator<<(ostream& s, const Color& c);

  int R;   // red value
  int G;   // green value
  int B;   // blue value

public:


  Color();    //  Default constructor sets all coordinates to zero.

  ///  Explicit constructor takes R, G, and B.
  Color(const int& r, const int& g, const int& b);

  Color& operator=(const Color& c);  //  Assignment operator.

  Color(const Color& c);      //  Copy constructor.

  ~Color();                   //  Destructor.

  /// Returns red value
  const int r(void) const;

  /// Returns green value
  const int g(void) const;

  /// Returns blue value
  const int b(void) const;

};

inline
istream& operator>>(istream& s, Color& c)
{
  int r;
  int g;
  int b;
  s >> r >> g >> b;
```

```
  c.R = r;
  c.G = g;
  c.B = b;
};

inline
ostream& operator<<(ostream& s, const Color& c)
{ s << (int)c.R << " " << (int)c.G << " " << (int)c.B << "\n"; };

/// Base class for translating a scalar value into a color.
class Colormap
{
private:

  Range<Float> Inputrange;
  Float CenterOfRange;
  Float DynamicRange;
  // this virtual member expects a scaled value from 0.0 to 1.0
  virtual const Color Map(const Float& f) const = 0;

public:

    //  Default constructor shouldn´t be called
  Colormap();

  ///  Explicit constructor takes the allowable input range.
  Colormap(const Range<Float> & therange, Float centerofrange = 0);

  //  Assignment operator.
  Colormap& operator=(const Colormap& cm);

    //  Copy constructor.
  Colormap(const Colormap& cm);

    //  Destructor.
  virtual ~Colormap();

  /// Converts a scalar to a color.
  /***
    Takes a value which should be within input range, truncates it to
    input range if necessary, computes the color and returns it.
   */
  const Color color(const Float& val) const;

  void setdynrange(const Float& val);

};

/// Color map transitioning from blue through green to red.
class ColormapBlueGreenRed : public Colormap
{

private:
```

```
  const Color Map(const Float& f) const;

public:

  ColormapBlueGreenRed();
  ColormapBlueGreenRed(const Range<Float> & therange);
  ~ColormapBlueGreenRed();
};

/// Hot metal color map going from black to red to yellow.
class ColormapBlackRedYellow : public Colormap
{

private:

  const Color Map(const Float& f) const;

public:

  ColormapBlackRedYellow();

  ColormapBlackRedYellow(const Range<Float> & therange);

  ~ColormapBlackRedYellow();

};

/// Cold color map going from black to blue to cyan.
class ColormapBlackBlueCyan : public Colormap
{

private:

  const Color Map(const Float& f) const;

public:

  ColormapBlackBlueCyan();

  ColormapBlackBlueCyan(const Range<Float> & therange);

  ~ColormapBlackBlueCyan();

};

// Monochrome color map from black to gray to white.
class ColormapBlackGrayWhite : public Colormap
{

private:

  const Color Map(const Float& f) const;

public:
```

```
  ColormapBlackGrayWhite();

  ColormapBlackGrayWhite(const Range<Float> & therange);

  ~ColormapBlackGrayWhite();

};

/// Color Doppler style color map with cold through hot colors.
class ColormapCyanBlueBlackRedYellow : public Colormap
{

private:

  const Color Map(const Float& f) const;

public:

  ColormapCyanBlueBlackRedYellow();

  ColormapCyanBlueBlackRedYellow(const Range<Float> & therange);

  ~ColormapCyanBlueBlackRedYellow();

};

#endif     /* !__Colormap_h */
```

B.3 Colormap.cc

```
#include "Colormap.h"
#include "cassert"

// The Color defs

Color::Color()
{
  R = 0;
  G = 0;
  B = 0;
};

Color::Color(const int& r, const int& g, const int& b)
{
  R = r;
  G = g;
  B = b;
};

Color& Color::operator=(const Color& c)
{
  R = c.r();
```

```
  G = c.g();
  B = c.b();
  return *this;
};

Color::Color(const Color& c)
{ *this = c; };

Color::~Color()
{ };

const int Color::r(void) const
{ return R; };

const int Color::g(void) const
{ return G; };

const int Color::b(void) const
{ return B; };

// The Colormap defs

Colormap::Colormap()
{ cerr << "Warning: calling default constructor for Colormap.\n"; };

Colormap::Colormap(const Range<Float> & therange, Float centerofrange = 0)
: Inputrange(therange), CenterOfRange(centerofrange), DynamicRange(1.0)
{ };

    //  Assignment operator.
Colormap& Colormap::operator=(const Colormap& cm)
{
  Inputrange = cm.Inputrange;
  CenterOfRange = cm.CenterOfRange;
  DynamicRange = cm.DynamicRange;
  return *this;
};

Colormap::Colormap(const Colormap& cm)
{ *this = cm; };

Colormap::~Colormap() {};

const Color Colormap::color(const Float& val) const
{
  Float f = Inputrange.truncate(val);
  f = ((f - CenterOfRange) / DynamicRange) + CenterOfRange;
  f = Inputrange.fraction(f);
  if (f > 1.0) f = 1.0;
  if (f < 0.0) f = 0.0;
  return Map(f);
};

void Colormap::setdynrange(const Float& val)
```

```
{
  DynamicRange = val;
  if (DynamicRange > 1.0) DynamicRange = 1.0;
  if (DynamicRange < 0.01) DynamicRange = 0.01;
};

const Color ColormapBlueGreenRed::Map(const Float& f) const
{
  int r = 0;
  int g = 0;
  int b = 0;
  if (f < 0.5)
    {
      g = (int)(f * 2 * 255.0);
      b = (int)((0.5 - f) * 2 * 255.0);
    }
  else
    {
      r = (int)((f - 0.5) * 2 * 255.0);
      g = (int)((1.0 - f) * 2 * 255.0);
    };
  Color tmpcolor(r, g, b);
  return tmpcolor;
};

ColormapBlueGreenRed::ColormapBlueGreenRed()
: Colormap()
{ };

ColormapBlueGreenRed::ColormapBlueGreenRed(const Range<Float> & therange)
: Colormap(therange)
{ };

ColormapBlueGreenRed::~ColormapBlueGreenRed()
{ };

const Color ColormapBlackRedYellow::Map(const Float& f) const
{
  int r = 0;
  int g = 0;
  int b = 0;
  if (f < 0.5)
    {
      r = (int)(f * 2 * 255.0);
    }
  else
    {
      r = 255;
      g = (int)((f - 0.5) * 2 * 255.0);
    };
  Color tmpcolor(r, g, b);
  return tmpcolor;
};
```

```
ColormapBlackRedYellow::ColormapBlackRedYellow()
: Colormap()
{ };

ColormapBlackRedYellow::ColormapBlackRedYellow(const Range<Float> & therange)
: Colormap(therange)
{ };

ColormapBlackRedYellow::~ColormapBlackRedYellow()
{ };

const Color ColormapBlackBlueCyan::Map(const Float& f) const
{
  int r = 0;
  int g = 0;
  int b = 0;
  if (f < 0.5)
    {
      b = (int)(f * 2 * 255.0);
    }
  else
    {
      g = (int)((f - 0.5) * 2 * 255.0);
      b = 255;
    };
  Color tmpcolor(r, g, b);
  return tmpcolor;
};

ColormapBlackBlueCyan::ColormapBlackBlueCyan()
: Colormap()
{ };

ColormapBlackBlueCyan::ColormapBlackBlueCyan(const Range<Float> & therange)
: Colormap(therange)
{ };

ColormapBlackBlueCyan::~ColormapBlackBlueCyan()
{ };

const Color ColormapBlackGrayWhite::Map(const Float& f) const
{
  int rgb = (int)(f * 255.0);
  Color tmpcolor(rgb, rgb, rgb);
  return tmpcolor;
};

ColormapBlackGrayWhite::ColormapBlackGrayWhite()
: Colormap()
{ };

ColormapBlackGrayWhite::ColormapBlackGrayWhite(const Range<Float> & therange)
: Colormap(therange)
{ };
```

```
ColormapBlackGrayWhite::~ColormapBlackGrayWhite()
{ };

const Color ColormapCyanBlueBlackRedYellow::Map(const Float& f) const
{
  int r = 0;
  int g = 0;
  int b = 0;
  assert(f >= -.00001 && f <= 1.00001);
  if (f < 0.25)
    {
      g = (int)((0.25 - f) * 2 * 2 * 255);
      b = 255;
    }
  else if (f < 0.5)
    {
      b = (int)((0.5 - f) * 2 * 2 * 255);
    }
  else if (f < 0.75)
    {
      r = (int)((f - 0.5) * 2 * 2 * 255);
    }
  else
    {
      r = 255;
      g = (int)((f - 0.75) * 2 * 2 * 255);
    };
  Color tmpcolor(r, g, b);
  return tmpcolor;
};

ColormapCyanBlueBlackRedYellow::ColormapCyanBlueBlackRedYellow()
: Colormap()
{ };

ColormapCyanBlueBlackRedYellow::ColormapCyanBlueBlackRedYellow(const Range<Float> & therange)
: Colormap(therange)
{ };

ColormapCyanBlueBlackRedYellow::~ColormapCyanBlueBlackRedYellow()
{ };
```

B.4 WTobj.h

```
#ifndef __WTobj_h
#define __WTobj_h

#include <cassert>
#include "Flowfield.h"
#include "utils.h"
#include "Colormap.h"
```

```
#include "wt.h"

/// Implementation of a graphical object.
/**
  This is currently used as a wrapper class for Sense8´s C library
  WorldToolKit.  The interface to this class should be applicable to
  other 3D graphics library.  The interface uses position and
  orientation classes from the Flowfield library, and the color class
  from the VirtualObject library.  The VirtualObject library should be
  portable to another platform/library just by rewriting this class
  implementation.  Note, the sensor object has not been encapsulated.
*/

class WTobj
{
private:

  WTobject* Obj;
  WTvertex** Verts;
  WTp3* Norms;
  int Numverts;
  int Nextvert;

public:
  /// Default constructor should be used.
  WTobj();

  /// Begin building object geometry, pass number of vertices.
  void openobject(const int vertexcount);

  /// Finish building geometry.
  void closeobject(void);

  /// Make the object a brick.
  void newblock(float x, float y, float z);

  /// Make the object a tesselated sphere.
  void newsphere(float radius, int nlat, int nlong);

  /// Make the object a tesellated cone.
  void newcone(float height, float radius, int tess);

  /// Make the object a tesellated cylinder.
  void newcylinder(float height, float radius, int tess);

  /// Link the object to a sensor.
  void addsensor(WTsensor* wts);

  /// Set the position of the object.
  void setposition(const Position& pos);

  /// Get the position of the object.
  const Position getposition(void);
```

```
  /// Set the orientation of the object.
  void setorientation(const Quaternion& ori);

  /// Get the orientation of the object.
  const Quaternion getorientation(void);

  /// Set the pivot point location.
  void setpivot(const Position& pos);

  /// Set the global color of the object.
  void setcolor(const Color& c);

  /// Sets the object to be visible (true) or invisible (false).
  void setvisibility(bool flag);

  /// Add a vertex to the object, it will have an index by insertion order.
  void addvertex(const Position& pos);

  /// Set the color of a vertex by its insertion number.
  void setvertexcolor(const int index, const Color& c);

  /// Get the color of a vertex by its insertion number.
  void getvertexcolor(const int index, Color& c);

  /// Set the outward normal vector of a vertex by its insertion number.
  void setvertexnormal(const int index, const Position& p);

  // Get the position of a vertex by its insertion number.
  void getvertexnormal(const int index, Position& p);

  /// Get the position of a vertex by its insertion number.
  const Position getvertexposition(int whichvertex);

  /// Add a Gouraud shaded polygon (triangle) by three vertices.
  void addpoly(const int vert1, const int vert2, const int vert3);

  /// Shift an object in its local reference frame.
  void translatelocal(const Position& deltap);

  /// Connect this object to another.
  void connectto(WTobj& obj);

  /// Blend the same-position vertex colors and normals together.
  void blend(void);

  /// Optimize structure of closed object for speed.
  void optimize(void);

  /// Return number of vertices in object.
  const int numverts();

};

#endif /* !__WTobj_h */
```

B.5 WTobj.cc

```
#include "WTobj.h"

WTobj::WTobj()
{
  Obj = 0;
  Verts = 0;
  Norms = 0;
  Numverts = 0;
  Nextvert = 0;
};

void WTobj::openobject(const int vertexcount)
{
  assert(vertexcount > 0);
  assert(Obj == 0);
  assert(Verts == 0);
  assert(Norms == 0);
  assert(Numverts == 0);
  assert(Nextvert == 0);
  Numverts = vertexcount;
  Verts = new (WTvertex*)[vertexcount];
  Norms = new WTp3[vertexcount];
  assert(Verts != 0);
  Obj = WTobject_begin();
  assert(Obj != 0);
};

void WTobj::closeobject(void)
{
  assert(Nextvert == Numverts); // we should have got them all.
  WTobject_close(Obj, FALSE, TRUE);
  WTobject_add(Obj);
  WTobject_setvisibility(Obj, FALSE);
};

void WTobj::newblock(float x, float y, float z)
{
  Obj = WTobject_newblock(x, y, z, FALSE, FALSE);
  assert(Obj != 0);
};

void WTobj::newsphere(float radius, int nlat, int nlong)
{
  Obj = WTobject_newsphere(radius, nlat, nlong, FALSE, TRUE, TRUE);
  assert(Obj != 0);
};

void WTobj::newcone(float height, float radius, int tess)
{
  Obj = WTobject_newcone(height, radius, tess, FALSE, TRUE);
  assert(Obj != 0);
};
```

```
void WTobj::newcylinder(float height, float radius, int tess)
{
  Obj = WTobject_newcylinder(height, radius, tess, FALSE, TRUE, TRUE);
  assert(Obj != 0);
};

void WTobj::addsensor(WTsensor* wts)
{
  WTobject_addsensor(Obj, wts, WTFRAME_VPOINT);
};

void WTobj::setposition(const Position& pos)
{
  WTp3 wtp;
  posconvert(pos, wtp);
  WTobject_setposition(Obj, wtp);
};

const Position WTobj::getposition(void)
{
  WTp3 wtp;
  Position p;
  WTobject_getposition(Obj, wtp);
  posconvert(wtp, p);
  return p;
};

const Quaternion WTobj::getorientation(void)
{
  WTq wtq;
  Quaternion q;
  WTobject_getorientation(Obj, wtq);
  oriconvert(wtq, q);
  return q;
};

void WTobj::setorientation(const Quaternion& ori)
{
  WTq wtq;
  oriconvert(ori, wtq);
  WTobject_setorientation(Obj, wtq);
};

void WTobj::setpivot(const Position& pos)
{
  WTp3 wtp;
  posconvert(pos,wtp);
  WTobject_setpivot(Obj,wtp,WTFRAME_LOCAL);
};

void WTobj::setcolor(const Color& c)
{
  WTobject_setrgb(Obj, (unsigned char)c.r(),
```

```
  (unsigned char)c.g(),
  (unsigned char)c.b());
};

void WTobj::setvisibility(bool flag)
{
  WTobject_setvisibility(Obj, flag);
};

void WTobj::addvertex(const Position& pos)
{
  assert(Nextvert < Numverts);
  WTp3 wtp;
  posconvert(pos, wtp);
  Verts[Nextvert] = WTvertex_new(wtp);
  assert(Verts[Nextvert] > 0);
  WTobject_addvertex(Obj, Verts[Nextvert]);
  Nextvert++;
};

void WTobj::setvertexcolor(const int index, const Color& c)
{
  assert(index >= 0 && index < Numverts && index < Nextvert);
  WTobject_setvertexrgb(Obj, Verts[index],
(unsigned short)c.r(),
(unsigned short)c.g(),
(unsigned short)c.b());
};

void WTobj::getvertexcolor(const int index, Color& c)
{
  assert(index >= 0 && index < Numverts && index < Nextvert);
  unsigned char r;
  unsigned char g;
  unsigned char b;
  WTobject_getvertexrgb(Obj, Verts[index], &r, &g, &b);
  Color ctmp(r,g,b);
  c = ctmp;
};

void WTobj::setvertexnormal(const int index, const Position& p)
{
  assert(index >= 0 && index < Numverts && index < Nextvert);
  WTp3 wtp;
  posconvert(p, wtp);
  WTobject_setvertexnormal(Obj, Verts[index], wtp);
  Norms[index] = wtp;
};

void WTobj::getvertexnormal(const int index, Position& p)
{
  assert(index >= 0 && index < Numverts && index < Nextvert);
  WTp3 wtp;
  /*
```

```
     This should work, but I think it is a buggy prototype since it
     doesn´t pass the position by pointer, so I store the normals
     manually in Norms[].
   */
  //  WTobject_getvertexnormal(Obj, Verts[index], wtp);
  wtp = Norms[index];
  posconvert(wtp,p);
};

const Position WTobj::getvertexposition(int whichvertex)
{
  assert(   whichvertex >= 0
 && whichvertex < Numverts
 && whichvertex < Nextvert);
  WTp3 wtp;
  WTobject_getvertexposition(Obj, Verts[whichvertex], wtp);
  Position p;
  posconvert(wtp, p);
  return p;
}

void WTobj::addpoly(const int vert1, const int vert2, const int vert3)
{
  assert(vert1 >= 0 && vert1 < Numverts && vert1 < Nextvert);
  assert(vert2 >= 0 && vert2 < Numverts && vert2 < Nextvert);
  assert(vert3 >= 0 && vert3 < Numverts && vert3 < Nextvert);
  assert(vert1 != vert2 && vert2 != vert3 && vert3 != vert1);
  WTpoly* poly = WTpoly_begin(Obj);
  WTpoly_addvertex(poly, vert1);
  WTpoly_addvertex(poly, vert2);
  WTpoly_addvertex(poly, vert3);
  WTpoly_close(poly);
  WTpoly_setbothsides(poly,TRUE);
};

void WTobj::translatelocal(const Position& deltap)
{
  WTp3 wtp;
  posconvert(deltap, wtp);
  WTpq wtpq;
  WTq wtq;
  wtq[0] = 0.0;
  wtq[1] = 0.0;
  wtq[2] = 0.0;
  wtq[3] = 1.0;
  wtpq.p = wtp;
  wtpq.q = wtq;
  WTobject_move(Obj, &wtpq, WTFRAME_LOCAL);
};

void WTobj::connectto(WTobj& obj)
{
  WTobject_attach(obj.Obj, Obj);
};
```

```
void WTobj::blend(void)
{
  const int numbuckets = 200;
  vector<list<int> > hashtable(numbuckets);
  int v;
  Position vertp;
  int x;  int y;  int z;  int hashindex;
  // loop through all vertices and hash them based on position
  for (v = 0; v < numverts(); v++)
    {
      vertp = getvertexposition(v);
      x = (int)(vertp.x()*263.0);
      y = (int)(vertp.y()*347.0);
      z = (int)(vertp.z()*443.0);
      hashindex = (x + y + z) % numbuckets;
      if (hashindex < 0) hashindex *= -1;
      hashtable[hashindex].push_back(v);
    };
  // now we have a vector (called hashtable) which contains lists of likely
  // position matching vertex indices.

  // first loop through the hashtable and grab each list
  for (unsigned int b = 0; b < hashtable.size(); b++)
    {
      list<int>& candlist = hashtable[b];  // just a ref for readability
      // we will hash again into a list of lists of real matches.
      list<list<int> > subhashtable;
      // loop through all vertices in candlist, put in same-position buckets.
      for (list<int>::iterator c = candlist.begin();
   c != candlist.end();
   c++)
{
  int& v = *c;
  Position vertp = getvertexposition(v);
  // search through subhashtable to find matching position
  bool found = false;
  for (list<list<int> >::iterator which = subhashtable.begin();
       !found && which != subhashtable.end();
       which++)
    {
      if (vertp == getvertexposition(*((*which).begin())))
// got a match
{
  // add vertex index to that bucket
  (*which).push_back(v);
  found = true;
};
    };
  // if we couldn´t find it, make a new bucket for this vertex
  if (!found)
    {
      list<int> newbucket;
      newbucket.push_back(v);
```

```
      subhashtable.push_back(newbucket);
    };
};
      // the bucket list should be complete, now we can process it.
      for (list<list<int> >::iterator whichbucket = subhashtable.begin();
   whichbucket != subhashtable.end();
   whichbucket++)
{
  int r = 0;
  int g = 0;
  int b = 0;
  Position norm(0,0,0);
  int numslots = (*whichbucket).size();
  for (list<int>::iterator whichslot = (*whichbucket).begin();
       whichslot != (*whichbucket).end();
       whichslot++)
    {
      Color thiscolor;
      getvertexcolor(*whichslot, thiscolor);
      r += thiscolor.r();
      g += thiscolor.g();
      b += thiscolor.b();
      Position thisnorm;
      getvertexnormal(*whichslot, thisnorm);
      norm = norm + thisnorm;
    };
  r /= numslots;
  g /= numslots;
  b /= numslots;
  Color blendcolor(r,g,b);
  norm = (1.0/numslots)*norm;
  for (list<int>::iterator whichslot = (*whichbucket).begin();
       whichslot != (*whichbucket).end();
       whichslot++)
    {
      setvertexcolor(*whichslot, blendcolor);
      setvertexnormal(*whichslot, norm);
    };
};
    };
};

void WTobj::optimize()
{
  WTobject_prebuild(Obj);
};

const int WTobj::numverts()
{
  return Numverts;
};
```

B.6 DynVar.h

```
#ifndef __DepVar_h
#define __DepVar_h


/// Dynamically links variable values to other variable values.
/**
  A dynamic variable provides a convenient way to connect the value
  of a code variable to another variable´s value at run time.  It can
  be in one of two states: static or dynamic.  If static, the DynVar
  keeps a copy of its data local, and is independent.  The value can
  be changed just like a regular variable, except it should be accessed
  using operator().  If dynamic, the DynVar is linked to another DynVar.
  When the data is accessed (using the same operator() calls) the linked
  DynVar is queried to get the real value.  It is possible that this
  DynVar will also be linked and recursion will continue until a static
  DynVar is reached.  An offset value can be set in a DynVar which will
  automatically be added to the linked value.

  Warning: this class does not check for cyclic graph structures.  If
  one of these is created and accessed, it will cause an infinite loop.
*/
template <class T>
class DynVar
{
private:

  T TheVar;
  T Delta;
  bool Hasdelta;
  DynVar<T>* Link;

public:

  DynVar();

  /// Construct and initialize value.
  DynVar(const T val);

  /// Construct with initial value and delta offset.
  DynVar(const T val, const T delta);

  DynVar<T>& operator=(const DynVar<T>& dv);

  DynVar(const DynVar<T>& dv);

  virtual ~DynVar();

  /// Lock the variable at its current value and unlink it.
  void unlink(void);

  /// Return the current value.
  const T operator()(void);
```

```
  /// Set the variable to a constant value and return reference to same.
  T& operator()(const T& t);

  /// Set value to a constant offset from the linked variable.
  void setdelta(const T& delta);

  /// Link to another dynamic variable.
  void link(DynVar<T>& l);

  /// Link to another dynamic variable, with a constant offset.
  void link(DynVar<T>& l, const T& delta);

  /// Check if the linked value has changed.
  const bool touched(void);

  /// Print a status report to standard error.
  void status(void);
};

// the definitions...

template<class T>
DynVar<T>::DynVar()
: Hasdelta(false), Link(0)
{};

template<class T>
DynVar<T>::DynVar(const T val)
: TheVar(val), Hasdelta(false), Link(0)
{};

template<class T>
DynVar<T>::DynVar(const T val, const T delta)
: TheVar(val), Delta(delta), Hasdelta(true), Link(0)
{};

template<class T>
DynVar<T>& DynVar<T>::operator=(const DynVar<T>& dv)
{
  if (this == &dv) return *this;
  TheVar = dv.TheVar;
  Delta = dv.Delta;
  Hasdelta = dv.Hasdelta;
  Link = dv.Link;
  return *this;
};

template<class T>
DynVar<T>::DynVar(const DynVar<T>& dv)
{ *this = dv; };

template<class T>
DynVar<T>::~DynVar()
```

```
{};

template<class T>
void DynVar<T>::unlink(void)
{
  Link = 0;
  Hasdelta = false;
};

template<class T>
const T DynVar<T>::operator()(void)
{
  if (Link != 0)
    {
      TheVar = Link->operator()();
      if (Hasdelta) return TheVar + Delta;
    };
  return TheVar;
};

template<class T>
T& DynVar<T>::operator()(const T& t)
{
  unlink();
  TheVar = t;
  return TheVar;
};

template<class T>
void DynVar<T>::setdelta(const T& delta)
{
  Delta = delta;
  Hasdelta = true;
};

template<class T>
void DynVar<T>::link(DynVar<T>& l)
{
  Link = &l;
  Hasdelta = false;
};

template<class T>
void DynVar<T>::link(DynVar<T>& l, const T& delta)
{
  Link = &l;
  Delta = delta;
  Hasdelta = true;
};

template<class T>
const bool DynVar<T>::touched(void)
{
  return Link && (TheVar != Link->operator()());
```

```
};

template<class T>
void DynVar<T>::status(void)
{
  // should somehow assert that we have a DAG or this thing could loop...
  cerr << "DynVar: Link = " << Link << flush;
  cerr << " , Value = " << TheVar << "\n" << flush;
  cerr << " , Delta = " << Delta << "\n" << flush;
  if (Link)
    {
      cerr << " which links to: " << flush;
      Link->status();
    };
  cerr << "\n" << flush;
};

#endif  __DepVar_h
```

B.7 DynVar.cc

```
#include "DynVar.h"
```

B.8 VirtualObject.h

```
#ifndef __VirtualObject_h
#define __VirtualObject_h

#include "DynVar.h"
#include "Flowfield.h"

/// Abstract representation of graphic object in some abstract field.
class VirtualObject
{
private:

  DynVar<bool> Active;
  DynVar<bool> Focus;
  DynVar<Position> Pos;
  DynVar<Quaternion> Ori;
  DynVar<Position> PosRef;
  DynVar<Quaternion> OriRef;
  DynVar<Position> PosRel;
  DynVar<Quaternion> OriRel;
  bool AbsRefFrame;
  Flowfield* Ff;

public:

  VirtualObject();

  VirtualObject(Flowfield& ff);
```

```
VirtualObject& operator=(const VirtualObject& v);

VirtualObject(const VirtualObject& v);

virtual ~VirtualObject();

/// Is the object on?
virtual bool active();

/// The update function should be called on every simulation loop.
virtual void update(int frame_count) = 0;

/// Turn on object.
virtual void activate(void);

/// Turn off object.
virtual void deactivate(void);

/// Turn on object focus.
virtual void focus(void);

/// Turn off object focus.
virtual void defocus(void);

/// Returns the position of the tool.
virtual Position position(void);

/// Returns the orientation of the tool.
virtual Quaternion orientation(void);

/// Sets the position of the tool.
virtual void setposition(const Position p);

/// Sets the orientation of the tool.
virtual void setorientation(const Quaternion q);

/// Link position relative to another tool in global space.
virtual void linktoabsposition(VirtualObject& fvt, const Position delta);

/// Link position relative to another tool in that tool´s local space.
virtual void linktorelposition(VirtualObject& fvt, const Position delta);

/// Free the tool´s position.
virtual void unlinkposition(void);

/// Link orientation to that of another tool.
virtual void linktoabsorientation(VirtualObject& fvt);

/// Free the tool´s orientation.
virtual void unlinkorientation(void);

/// Return flowfield reference
Flowfield& ff(void);
```

```
};

#endif /* !__VirtualObject_h */
```

B.9 VirtualObject.cc

```
#include "VirtualObject.h"

VirtualObject::VirtualObject()
: Active(false), Focus(false)
{};

VirtualObject::VirtualObject(Flowfield& ff)
{
  Ff = &ff;
  Pos(Position(0.0,0.0,0.0));
  Ori(Quaternion(0,Position(0,0,1)));
  PosRef(Position(0.0,0.0,0.0));
  OriRef(Quaternion(0,Position(0,0,1)));
  PosRel(Position(0.0,0.0,0.0));
  OriRel(Quaternion(0,Position(0,0,1)));
};

VirtualObject& VirtualObject::operator=(const VirtualObject& v)
{
  if (this == &v) return *this;
  Active = v.Active;
  Focus = v.Focus;
  return *this;
};

VirtualObject::VirtualObject(const VirtualObject& v)
{
  *this = v;
};

VirtualObject::~VirtualObject()
{};

bool VirtualObject::active()
{
  return Active();
};

void VirtualObject::activate(void)
{
  Active(true);
};

void VirtualObject::deactivate(void)
{
  Active(false);
};
```

```
void VirtualObject::focus(void)
{
  Focus(true);
};

void VirtualObject::defocus(void)
{
  Focus(false);
};

Quaternion VirtualObject::orientation(void)
{
  return Ori();
};

Position VirtualObject::position(void)
{
  if (!AbsRefFrame)
    {
      Quaternion q = OriRef();
      Pos(PosRel()*q.inverse() + PosRef());
    };
  return Pos();
};

void VirtualObject::setposition(const Position p)
{
  Pos(p);
  AbsRefFrame = true;
};

void VirtualObject::setorientation(const Quaternion q)
{
  Ori(q);
};

void VirtualObject::linktoabsposition(VirtualObject& fvt, const Position delta)
{
  Pos.link(fvt.Pos, delta);
  AbsRefFrame = true;
};

void VirtualObject::linktorelposition(VirtualObject& fvt, const Position delta)
{
  PosRel(delta);
  PosRef.link(fvt.Pos);
  OriRef.link(fvt.Ori);
  AbsRefFrame = false;
};

void VirtualObject::unlinkposition(void)
{
  Pos(position());
```

```
  AbsRefFrame = true;
};

void VirtualObject::linktoabsorientation(VirtualObject& fvt)
{
  Ori.link(fvt.Ori);
};

void VirtualObject::unlinkorientation(void)
{
  Ori.unlink();
};

Flowfield& VirtualObject::ff(void)
{
  return *Ff;
};
```

B.10 Widget.h

```
#ifndef __Widget_h
#define __Widget_h

#include "VirtualObject.h"
class Widget;  // fwd declaration
#include "FlowVisTool.h"
#include "Colormap.h"
#include "Hand.h"
#include "WTobj.h"

/// Abstract representation of a control handle.
class Widget : public VirtualObject
{
protected:

  FlowVisTool* Owner;
  bool Grabbed;
  Position Relpos;
  WTobj Shape;

public:

  Widget();
  /// Constructor takes tool Widget connects to, and its relative position.
  Widget(FlowVisTool& owner, Position relpos);
  Widget& operator=(const Widget& w);
  Widget(const Widget& w);
  virtual ~Widget();
  /// Call this on every simulation loop.
  virtual void update(int /* frame_count */) = 0;
  /// Is it grabbed by a hand?
  virtual bool grabbed(void);
```

```
  /// Grab the object by a hand.
  virtual void grabbedby(Hand& h) = 0;
  /// Release the grab.
  virtual void ungrabbed(void) = 0;
  virtual void hilite(void) = 0;
  virtual void lolite(void) = 0;
};



/// Handle for controlling position.
class PosWidget : public Widget
{

public:

  PosWidget();

  /// Constructor takes tool Widget connects to, and its relative position.
  PosWidget(FlowVisTool& owner, Position relpos);

  PosWidget& operator=(const PosWidget& pw);

  PosWidget(const PosWidget& pw);

  virtual ~PosWidget();

  /// This should be called on every simulation loop.
  virtual void update(int /* frame_count */);

  /// Grab the object by a hand.
  virtual void grabbedby(Hand& h);

  /// Release the grab.
  virtual void ungrabbed(void);

  virtual void hilite(void);

  virtual void lolite(void);

  /// Turn it on.
  virtual void activate(void);

  /// Turn it off.
  virtual void deactivate(void);

};

/// Handle for controlling orientation.
class OriWidget : public Widget
{

public:
  OriWidget();
```

```
  /// Constructor takes tool Widget connects to, and its relative position.
  OriWidget(FlowVisTool& owner, Position relpos);

  OriWidget& operator=(const OriWidget& ow);

  OriWidget(const OriWidget& ow);

  virtual ~OriWidget();

  /// This should be called on every simulation loop.
  virtual void update(int /* frame_count */);

  /// Grab the object by a hand.
  virtual void grabbedby(Hand& h);

  /// Release the grab.
  virtual void ungrabbed(void);

  /// Turn it on.
  virtual void hilite(void);

  virtual void lolite(void);

  virtual void activate(void);

  /// Turn it off.
  virtual void deactivate(void);

};

/// Handle for controlling dynamic range
class DynRangeWidget : public Widget
{
private:

WTobj Shape;
Hand* H;

public:

  DynRangeWidget();

  /// Constructor takes tool Widget connects to, and its relative position.
  DynRangeWidget(FlowVisTool& owner, Position relpos);

  DynRangeWidget& operator=(const DynRangeWidget& pw);

  DynRangeWidget(const DynRangeWidget& pw);

  virtual ~DynRangeWidget();

  /// This should be called on every simulation loop.
  virtual void update(int /* frame_count */);
```

```
  /// Grab the object by a hand.
  virtual void grabbedby(Hand& h);

  /// Release the grab.
  virtual void ungrabbed(void);

  virtual void hilite(void);

  virtual void lolite(void);

  /// Turn it on.
  virtual void activate(void);

  /// Turn it off.
  virtual void deactivate(void);

};

#endif /* !__Widget_h */
```

B.11 Widget.cc

```
#include "Widget.h"

Widget::Widget()
{};

Widget::Widget(FlowVisTool& owner, Position relpos)
{
  Relpos = relpos;
  Owner = &owner;
  Grabbed = false;
};

Widget& Widget::operator=(const Widget& w)
{
  if (this == &w) return *this;
  VirtualObject::operator=(w);
  Owner = w.Owner;
  Grabbed = w.Grabbed;
  Relpos = w.Relpos;
  Shape = w.Shape;
  return *this;
};

Widget::Widget(const Widget& w)
{
  *this = w;
};

Widget::~Widget()
{};
```

```
bool Widget::grabbed(void)
{
  return Grabbed;
};

PosWidget::PosWidget()
{};

PosWidget::PosWidget(FlowVisTool& owner, Position relpos)
: Widget(owner, relpos)
{
  Shape.newblock(0.4, 0.4, 0.4);
  lolite();
  Shape.setvisibility(false);
};

PosWidget& PosWidget::operator=(const PosWidget& pw)
{
  if (this == &pw) return *this;
  Widget::operator=(pw);
  return *this;
};

PosWidget::PosWidget(const PosWidget& pw)
{
  *this = pw;
};

PosWidget::~PosWidget()
{};

void PosWidget::update(int /* frame_count */)
{
  Shape.setposition(position());
};

void PosWidget::grabbedby(Hand& h)
{
  if (!Grabbed)
    {
      hilite();
      linktoabsposition(h, Position(0,0,0));
      Owner->linktorelposition(*this,
  (-Relpos)*(Owner->orientation().inverse()));
      Grabbed = true;
    };
};

void PosWidget::ungrabbed(void)
{
  if (Grabbed)
    {
      lolite();
```

```
      Owner->unlinkposition();
      linktorelposition(*Owner, Relpos);
      Grabbed = false;
    };
};

void PosWidget::hilite(void)
{
  Shape.setcolor(Color(0xff, 0x70, 0x70));
};

void PosWidget::lolite(void)
{
  Shape.setcolor(Color(0xff, 0x00, 0x00));
};

void PosWidget::activate(void)
{
  VirtualObject::activate();
  Shape.setvisibility(true);
};

void PosWidget::deactivate(void)
{
  VirtualObject::deactivate();
  Shape.setvisibility(false);
};

OriWidget::OriWidget()
{};

OriWidget::OriWidget(FlowVisTool& owner, Position relpos)
: Widget(owner, relpos)
{
  Shape.newsphere(0.25, 8, 8);
  lolite();
  Shape.setvisibility(false);
};

OriWidget& OriWidget::operator=(const OriWidget& ow)
{
  if (this == &ow) return *this;
  Widget::operator=(ow);
  return *this;
};

OriWidget::OriWidget(const OriWidget& ow)
{
  *this = ow;
};

OriWidget::~OriWidget()
{};
```

```
void OriWidget::update(int /* frame_count */)
{
  Shape.setposition(position());
  Shape.setorientation(orientation());
};

void OriWidget::grabbedby(Hand& h)
{
  hilite();
  linktoabsorientation(h);
  Owner->linktoabsorientation(*this);
  Grabbed = true;
};

void OriWidget::ungrabbed(void)
{
  lolite();
  Owner->unlinkorientation();
  unlinkorientation();
  Grabbed = false;
};

void OriWidget::hilite(void)
{
  Shape.setcolor(Color(0x70, 0x70, 0xff));
};

void OriWidget::lolite(void)
{
  Shape.setcolor(Color(0x00, 0x00, 0xff));
};


void OriWidget::activate(void)
{
  VirtualObject::activate();
  Shape.setvisibility(true);
};

void OriWidget::deactivate(void)
{
  VirtualObject::deactivate();
  Shape.setvisibility(false);
};

DynRangeWidget::DynRangeWidget()
{};

DynRangeWidget::DynRangeWidget(FlowVisTool& owner, Position relpos)
: Widget(owner, relpos)
{
  Shape.newcylinder(0.5, 0.25, 16);
  lolite();
  Shape.setvisibility(false);
```

```
};

DynRangeWidget& DynRangeWidget::operator=(const DynRangeWidget& pw)
{
  if (this == &pw) return *this;
  Widget::operator=(pw);
  return *this;
};

DynRangeWidget::DynRangeWidget(const DynRangeWidget& pw)
{
  *this = pw;
};

DynRangeWidget::~DynRangeWidget()
{};

void DynRangeWidget::update(int /* frame_count */)
{
  Shape.setposition(position());
  if (Grabbed)
    {
      Float xdist = H->getposition().x() - Shape.getposition().x();
      Float ydist = H->getposition().y() - Shape.getposition().y();
      Float zdist = H->getposition().z() - Shape.getposition().z();
      Float val = (sqrt(xdist*xdist+ydist*ydist+zdist*zdist))/5.0;
      val = val*val;
      Owner->cmap().setdynrange(val);
    };
};

void DynRangeWidget::grabbedby(Hand& h)
{
  if (!Grabbed)
    {
      hilite();
      H = &h;
      Grabbed = true;
    };
};

void DynRangeWidget::ungrabbed(void)
{
  if (Grabbed)
    {
      lolite();
      Grabbed = false;
    };
};

void DynRangeWidget::hilite(void)
{
  Shape.setcolor(Color(0x70, 0xff, 0x70));
};
```

```
void DynRangeWidget::lolite(void)
{
  Shape.setcolor(Color(0x00, 0xff, 0x00));
};

void DynRangeWidget::activate(void)
{
  VirtualObject::activate();
  Shape.setvisibility(true);
};

void DynRangeWidget::deactivate(void)
{
  VirtualObject::deactivate();
  Shape.setvisibility(false);
};
```

B.12 FlowVisTool.h

```
#ifndef __FlowVisTool_h
#define __FlowVisTool_h

#include "Flowfield.h"
#include "VirtualObject.h"
#include <list.h>
class FlowVisTool; // fwd declaration
#include "Widget.h"
#include "Colormap.h"

/// An abstract tool for visualizing 3D transient fluid flow.
class FlowVisTool : public VirtualObject
{
private:

  list<Widget*> Widglist;

protected:

  int Last_frame_count;

public:
  FlowVisTool();

  // Constructor takes the flow field.
  FlowVisTool(Flowfield& ff);

  FlowVisTool& operator=(const FlowVisTool& fvt);

  FlowVisTool(const FlowVisTool& fvt);

  virtual ~FlowVisTool();
```

```
  void addwidget(Widget* wp);

  void activatewidgets(void);

  void deactivatewidgets(void);

  list<Widget*>& widgetlist(void);

  virtual Colormap& cmap(void);

};

#endif /* !__FlowVisTool_h */
```

B.13 FlowVisTool.cc

```
#include "FlowVisTool.h"

FlowVisTool::FlowVisTool()
: VirtualObject(), Last_frame_count(-1)
{};

FlowVisTool::FlowVisTool(Flowfield& ff)
: VirtualObject(ff), Last_frame_count(-1)
{
};

FlowVisTool& FlowVisTool::operator=(const FlowVisTool& fvt)
{
  if (this == &fvt) return *this;
  VirtualObject::operator=(fvt);
  Last_frame_count = fvt.Last_frame_count;
  return *this;
};

FlowVisTool::FlowVisTool(const FlowVisTool& fvt)
{
  *this = fvt;
};

FlowVisTool::~FlowVisTool()
{};

void FlowVisTool::addwidget(Widget* wp)
{
  assert(wp != 0);
  Widglist.push_back(wp);
};

void FlowVisTool::activatewidgets(void)
{
  list<Widget*>::iterator where = Widglist.begin();
  while (where != Widglist.end())
```

```
    {
      Widget* wp = *where++;
      wp->activate();
    };
};

void FlowVisTool::deactivatewidgets(void)
{
  list<Widget*>::iterator where = Widglist.begin();
  while (where != Widglist.end())
    {
      Widget* wp = *where++;
      wp->deactivate();
    };
};

list<Widget*>& FlowVisTool::widgetlist(void)
{
  return Widglist;
};

Colormap& FlowVisTool::cmap(void)
{ cerr << "oops!"; };
```

B.14 Hand.h

```
#ifndef __Hand_h
#define __Hand_h

#include "Flowfield.h"
#include "utils.h"
#include "DynVar.h"
#include "VirtualObject.h"
#include "WTobj.h"
class Hand;  // fwd declaration
#include "Widget.h"
#include "wt.h"

/// A hand in the virtual world for manipulating objects.
class Hand : public VirtualObject, private WTobj
{
private:

  Widget* Grabbedwidg;

public:

  /// Constructor takes device pointer for controlling hand.
  Hand(WTsensor* input_device);       // not good - interface dep´t on WT

  void centerinvp(WTviewpoint* wtvp);  // not good - interface dep´t on WT

  Hand& operator=(const Hand& h);
```

```
  Hand(const Hand& h);

  virtual ~Hand();

  /// Update function should be called for each frame of simulation.
  void update(int /* frame_count */);

  bool isgrabbing(void);

  bool cangrab(Widget* w);

  void grab(Widget* w);

  Widget* grabbed(void);

  void ungrab(void);

  const Position getposition(void);

};

#endif /* !__Hand_h */
```

B.15 Hand.cc

```
#include "Hand.h"

Hand::Hand(WTsensor* input_device)
{
  Grabbedwidg = 0;
  newcone(1.0, 0.15, 24);
  setcolor(Color(0xff, 0xff, 0));
  addsensor(input_device);
  activate();
};

void Hand::centerinvp(WTviewpoint* wtvp)
{
  WTviewpoint* tempvp = WTviewpoint_copy(wtvp);
  WTp3 trans;
  trans[2] = 10;
  WTviewpoint_translate(tempvp, trans, WTFRAME_LOCAL);
  WTp3 newp;
  WTviewpoint_getposition(tempvp, newp);
  Position newpos;
  posconvert(newp, newpos);
  WTobj::setposition(newpos);
  WTviewpoint_delete(tempvp);
};

Hand& Hand::operator=(const Hand& h)
{
```

```
  if (this == &h) return *this;
  VirtualObject::operator=(h);
  Grabbedwidg = h.Grabbedwidg;
  return *this;
};

Hand::Hand(const Hand& h)
{
  *this = h;
};

Hand::~Hand()
{};

void Hand::update(int /* frame_count */)
{
  VirtualObject::setposition(WTobj::getposition());
  VirtualObject::setorientation(WTobj::getorientation());
};

bool Hand::isgrabbing(void)
{
  if (Grabbedwidg) return true;
  return false;
};

bool Hand::cangrab(Widget* w)
{
  if (!w->active()) return false;
  Position handpos = position();
  Position widgpos = w->position();
  Position between = handpos - widgpos;
  Float dist = between.magnitude();
  return (dist < 1.0);
};

void Hand::grab(Widget* w)
{
  assert(Grabbedwidg == 0);
  Grabbedwidg = w;
  Grabbedwidg->grabbedby(*this);
};

Widget* Hand::grabbed(void)
{
  return Grabbedwidg;
};

void Hand::ungrab(void)
{
  assert(Grabbedwidg != 0);
  Grabbedwidg->ungrabbed();
  Grabbedwidg = 0;
};
```

```
const Position Hand::getposition(void)
{
  return WTobj::getposition();
};
```

B.16 StatObjTool.h

```
#ifndef __StatObjTool_h
#define __StatObjTool_h

#include "utils.h"
#include "Flowfield.h"
#include "VirtualObject.h"
#include "FlowVisTool.h"
#include "Widget.h"
#include "DynVar.h"
#include "Colormap.h"
#include "WTobj.h"

/// Abstract representation of tool with changing colors and static geometry.
class StatObjTool : public FlowVisTool
{
protected:

  Colormap* Cm;
  WTobj Shape;
  virtual const Float Valueof(const PosTimestep& pt) = 0;
  virtual const Color Colorof(const PosTimestep& pt);

public:

  StatObjTool();

  /// Constructor takes flow field, color map, and local positions of widgets.
  StatObjTool(Flowfield& ff, Colormap& cmap,
      Position prelpos, Position orelpos);

  StatObjTool& operator=(const StatObjTool& cp);

  StatObjTool(const StatObjTool& cp);

  virtual ~StatObjTool(void);

  /// Call update every simulation loop.
  virtual void update(int frame_count);

  /// Turn on tool.
  virtual void activate(void);

  /// Turn off tool.
  virtual void deactivate(void);
```

```
  Colormap& cmap(void) { return *Cm; };

};

#endif /* !__StatObjTool_h */
```

B.17 StatObjTool.cc

```
#include "StatObjTool.h"

const Color StatObjTool::Colorof(const PosTimestep& pt)
{
  Float val = Valueof(pt);
  Color clr = Cm->color(val);
  return clr;
};

StatObjTool::StatObjTool()
: FlowVisTool(), Cm(0)
{};

StatObjTool::StatObjTool(Flowfield& ff, Colormap& cmap,
 Position prelpos, Position orelpos)
: FlowVisTool(ff), Cm(&cmap)
{

  // make and link up the poswidget
  PosWidget* posw = new PosWidget(*this,prelpos);
  assert(posw != 0);
  addwidget(posw);
  posw->linktorelposition(*this, prelpos);

  // make and link up the oriwidget
  // link the oriwidget position back to the poswidget
  OriWidget* oriw = new OriWidget(*this, orelpos);
  assert(oriw != 0);
  addwidget(oriw);
  oriw->linktorelposition(*this, orelpos);
};

StatObjTool& StatObjTool::operator=(const StatObjTool& sot)
{
  if (this == &sot) return *this;
  FlowVisTool::operator=(sot);
  Cm = sot.Cm;
  Shape = sot.Shape;
};

StatObjTool::StatObjTool(const StatObjTool& sot)
{
  *this = sot;
};
```

```
StatObjTool::~StatObjTool(void)
{
};

void StatObjTool::update(int frame_count)
{
  // check if we need to update

  if (active())
    {
      Last_frame_count = frame_count;

      // first move/rotate the whole object
      Shape.setorientation(orientation());
      Shape.setposition(position());

      // then update all of the colors
      for (int vert = 0; vert < Shape.numverts(); vert++)
{
  Position p = Shape.getvertexposition(vert);
  PosTimestep pt(p, frame_count);
  Color c = Colorof(pt);
  Shape.setvertexcolor(vert, c);
};
    };
};

void StatObjTool::activate(void)
{
  FlowVisTool::activate();
  Shape.setvisibility(true);
  activatewidgets();
};

void StatObjTool::deactivate(void)
{
  FlowVisTool::deactivate();
  Shape.setvisibility(false);
  deactivatewidgets();
};
```

B.18 Cuttingplane.h

```
#ifndef __Cuttingplane_h
#define __Cuttingplane_h

#include "StatObjTool.h"

/// Rectangular plane display color mapped flow info.
class Cuttingplane : public StatObjTool
{
protected:
```

```
  DynVar<int> X;          // # of tiles in "X" direction
  DynVar<int> Y;          // # of tiles in "Y" direction
  DynVar<int> Xper1;      // number of tiles per 1 x unit
  DynVar<int> Yper1;      // number of tiles per 1 y unit
  virtual const Float Valueof(const PosTimestep& pt) = 0;
  WTobj FrameSides[4];
  DynRangeWidget* Dynw;

public:

  Cuttingplane();

  /// Constructor takes the flowfield and the colormap.
  Cuttingplane(Flowfield& ff, Colormap& cmap);

  Cuttingplane& operator=(const Cuttingplane& cp);

  Cuttingplane(const Cuttingplane& cp);

  virtual ~Cuttingplane(void);

  /// Turn on tool.
  virtual void activate(void);

  /// Turn off tool.
  virtual void deactivate(void);

};

/// Cutting plane displaying absolute speed on the surface.
class SpeedCuttingplane : public Cuttingplane
{
protected:

  virtual const Float Valueof(const PosTimestep& pt);

public:

  SpeedCuttingplane();

  /// Constructor takes the flowfield and the colormap.
  SpeedCuttingplane(Flowfield& ff, Colormap& cm);

  SpeedCuttingplane& operator=(const SpeedCuttingplane& cp);

  SpeedCuttingplane(const SpeedCuttingplane& scp);

  virtual ~SpeedCuttingplane(void);

};

/// Cutting plane displaying absolute strain rate on the surface.
class AbsStrainRateCuttingplane : public Cuttingplane
```

```
{
protected:

  virtual const Float Valueof(const PosTimestep& pt);

public:

  AbsStrainRateCuttingplane();

  /// Constructor takes the flowfield and the colormap.
  AbsStrainRateCuttingplane(Flowfield& ff, Colormap& cm);

  AbsStrainRateCuttingplane& operator=(const AbsStrainRateCuttingplane& cp);

  AbsStrainRateCuttingplane(const AbsStrainRateCuttingplane& asrcp);

  virtual ~AbsStrainRateCuttingplane(void);

};


/// Cutting plane displaying speed normal to the surface.
class NormSpeedCuttingplane : public Cuttingplane
{
private:

  WTobj FrameArrows[4];

protected:

  virtual const Float Valueof(const PosTimestep& pt);

public:

  NormSpeedCuttingplane();

  /// Constructor takes the flowfield and the colormap.
  NormSpeedCuttingplane(Flowfield& ff, Colormap& cm);

  NormSpeedCuttingplane& operator=(const NormSpeedCuttingplane& cp);

  NormSpeedCuttingplane(const NormSpeedCuttingplane& scp);

  virtual ~NormSpeedCuttingplane(void);

  /// Turn on tool.
  virtual void activate(void);

  /// Turn off tool.
  virtual void deactivate(void);

};

#endif /* !__Cuttingplane_h */
```

B.19 Cuttingplane.cc

```
#include "Cuttingplane.h"

Cuttingplane::Cuttingplane()
{};

Cuttingplane::Cuttingplane(Flowfield& ff, Colormap& cmap)
: StatObjTool(ff,cmap,Position(-2.5,2.5,0.0),Position(2.5,2.5,0.0)),
  X(15), Y(15), Xper1(3), Yper1(3)
{
  // begin creating the object
  Shape.openobject((X() + 1)*(Y() + 1));

  // add the vertices
  for (int j = 0; j <= Y(); j++)
    for (int i = 0; i <= X(); i++)
      {
int thisindex = j*(X() + 1) + i;
Shape.addvertex(Position((Float)i/Xper1(), (Float)j/Yper1(), 0.0));
Shape.setvertexcolor(thisindex, Color(0,0,0));
Shape.setvertexnormal(thisindex, Position(0.0, 0.0, 1.0));
      };

  // add the polygons
  for (int j = 0; j < Y(); j++)
    for (int i = 0; i < X(); i++)
      {
/*
 *   example of vertex numbering and polygon drawing:
 *  for X=Y=2;
 *  for j=i=1; 4-5-7 and 8-7-5
 *  6 7 8
 *  3 4 5
 *  0 1 2
 */
// the lower triangle
Shape.addpoly(i + j*(X() + 1),           //Lower Left
      i + j*(X() + 1) + 1,       //Lower Right
      i + (j + 1)*(X() + 1));    //Upper Left

// the upper triangle
Shape.addpoly(i + (j + 1)*(X() + 1) + 1, //Upper Right
      i + (j + 1)*(X() + 1),     //Upper Left
      i + j*(X() + 1) + 1);      //Lower Right
      };
  Shape.closeobject();

  // make frame
  for (int i = 0; i < 4; i++)
    {
      FrameSides[i].newcylinder(5, 0.03, 10);
      FrameSides[i].setvisibility(false);
    };
```

```
  FrameSides[0].setposition(Position(2.5,0,0));
  // The 1.0 passed to the Quaternion here is a hack.. I think it should
  // be PI/2 or something but this does not work.  Probably something
  // wierd with WTK...
  FrameSides[0].setorientation(Quaternion(1.0, Position(0,0,1)));
  FrameSides[1].setposition(Position(5,2.5,0));
  FrameSides[2].setposition(Position(2.5,5,0));
  // same hack here...
  FrameSides[2].setorientation(Quaternion(1.0, Position(0,0,1)));
  FrameSides[3].setposition(Position(0,2.5,0));
  for (int i = 0; i < 4; i++)
    {
      FrameSides[i].connectto(Shape);
    };

  Position drelpos(2.5,-2.5,0.0);
  Dynw = new DynRangeWidget(*this,drelpos);
  assert(Dynw != 0);
  addwidget(Dynw);
  Dynw->linktorelposition(*this, drelpos);

};

Cuttingplane& Cuttingplane::operator=(const Cuttingplane& cp)
{
  if (this == &cp) return *this;
  StatObjTool::operator=(cp);
  Cm = cp.Cm;
  X = cp.X;
  Y = cp.Y;
  Xper1 = cp.Xper1;
  Yper1 = cp.Yper1;
};

Cuttingplane::Cuttingplane(const Cuttingplane& cp)
{
  *this = cp;
};

Cuttingplane::~Cuttingplane(void)
{
};

void Cuttingplane::activate(void)
{
  for (int i = 0; i < 4; i++)
    {
      FrameSides[i].setvisibility(true);
    };
  StatObjTool::activate();
};

void Cuttingplane::deactivate(void)
{
```

```
  for (int i = 0; i < 4; i++)
    {
      FrameSides[i].setvisibility(false);
    };
  StatObjTool::deactivate();
};

const Float SpeedCuttingplane::Valueof(const PosTimestep& pt)
{
  Flowpoint fp = ff().eval(pt);
  Float spd = fp.speed();
  return spd;
};

SpeedCuttingplane::SpeedCuttingplane()
: Cuttingplane()
{};

SpeedCuttingplane::SpeedCuttingplane(Flowfield& ff, Colormap& cm)
: Cuttingplane(ff, cm)
{};

SpeedCuttingplane& SpeedCuttingplane::operator=(const SpeedCuttingplane& cp)
{
  if (this == &cp) return *this;
  Cuttingplane::operator=(cp);
  return *this;
};

SpeedCuttingplane::SpeedCuttingplane(const SpeedCuttingplane& scp)
{
  *this = scp;
};

SpeedCuttingplane::~SpeedCuttingplane(void)
{ };

//
const Float AbsStrainRateCuttingplane::Valueof(const PosTimestep& pt)
{
  Flowpoint fp = ff().eval(pt);
  Float spd = fp.absstrainrate();
  return spd;
};

AbsStrainRateCuttingplane::AbsStrainRateCuttingplane()
: Cuttingplane()
{};

AbsStrainRateCuttingplane::AbsStrainRateCuttingplane(Flowfield& ff,
     Colormap& cm)
: Cuttingplane(ff, cm)
{};
```

```
AbsStrainRateCuttingplane&
AbsStrainRateCuttingplane::operator=(const AbsStrainRateCuttingplane& cp)
{
  if (this == &cp) return *this;
  Cuttingplane::operator=(cp);
  return *this;
};

AbsStrainRateCuttingplane::AbsStrainRateCuttingplane(
  const AbsStrainRateCuttingplane& asrcp)
{
  *this = asrcp;
};

AbsStrainRateCuttingplane::~AbsStrainRateCuttingplane(void)
{ };

const Float NormSpeedCuttingplane::Valueof(const PosTimestep& pt)
{
  Flowpoint fp = ff().eval(pt);
  Quaternion q = orientation();
  Float spd = fp.speed(q);
  return spd;
};

NormSpeedCuttingplane::NormSpeedCuttingplane()
: Cuttingplane()
{};

NormSpeedCuttingplane::NormSpeedCuttingplane(Flowfield& ff, Colormap& cm)
: Cuttingplane(ff, cm)
{
  FrameArrows[0].openobject(3);
  FrameArrows[0].addvertex(Position(0,2,0));
  FrameArrows[0].addvertex(Position(0,3,0));
  FrameArrows[0].addvertex(Position(0,2.5,0.25));
  FrameArrows[0].addpoly(0,1,2);
  FrameArrows[0].closeobject();
  FrameArrows[0].setvisibility(false);
  FrameArrows[0].connectto(Shape);

  FrameArrows[1].openobject(3);
  FrameArrows[1].addvertex(Position(5,2,0));
  FrameArrows[1].addvertex(Position(5,3,0));
  FrameArrows[1].addvertex(Position(5,2.5,0.25));
  FrameArrows[1].addpoly(0,1,2);
  FrameArrows[1].closeobject();
  FrameArrows[1].setvisibility(false);
  FrameArrows[1].connectto(Shape);

  FrameArrows[2].openobject(3);
  FrameArrows[2].addvertex(Position(2,0,0));
  FrameArrows[2].addvertex(Position(3,0,0));
  FrameArrows[2].addvertex(Position(2.5,0,0.25));
```

```
  FrameArrows[2].addpoly(0,1,2);
  FrameArrows[2].closeobject();
  FrameArrows[2].setvisibility(false);
  FrameArrows[2].connectto(Shape);

  FrameArrows[3].openobject(3);
  FrameArrows[3].addvertex(Position(2,5,0));
  FrameArrows[3].addvertex(Position(3,5,0));
  FrameArrows[3].addvertex(Position(2.5,5,0.25));
  FrameArrows[3].addpoly(0,1,2);
  FrameArrows[3].closeobject();
  FrameArrows[3].setvisibility(false);
  FrameArrows[3].connectto(Shape);

};

NormSpeedCuttingplane&
NormSpeedCuttingplane::operator=(const NormSpeedCuttingplane& cp)
{
  if (this == &cp) return *this;
  Cuttingplane::operator=(cp);
  return *this;
};

NormSpeedCuttingplane::NormSpeedCuttingplane(const NormSpeedCuttingplane& scp)
{
  *this = scp;
};

NormSpeedCuttingplane::~NormSpeedCuttingplane(void)
{ };


void NormSpeedCuttingplane::activate(void)
{
  for (int i = 0; i < 4; i++) FrameArrows[i].setvisibility(true);
  Cuttingplane::activate();
};

void NormSpeedCuttingplane::deactivate(void)
{
  for (int i = 0; i < 4; i++) FrameArrows[i].setvisibility(false);
  Cuttingplane::deactivate();
};
```

B.20 Doppler.h

```
#ifndef __Doppler_h
#define __Doppler_h

#include <iostream.h>
#include <fstream.h>
```

```
#include "StatObjTool.h"

/// A simulation of the Cardiometrics flowire Doppler probe.
/**
  Models an endovascular Doppler probe which measures velocities in a
  conical volume.  The probe measures flow value ranges in the cone.  A
  geometric representation of the cone is displayed, and flow values are
  displayed on the surface.  Constructs and manipulates the mathematical
  model of the cone and measures samples within it at each timestep.
*/

class Doppler : public StatObjTool {

protected:

  Float Halfangle;    //  Half-angle of the sample cone in radians.
  Float Height;       //  Height of the sample cone.
  int Z_samples;      //  Samples in axial direction.
  int R_samples;      //  Samples in radial direction.
  int C_samples;      //  Samples in circumferential direction.

  // Returns position in static local coordinate system of
  // sample point given the indices in the Z direction, R_adial
  // direction, and C_ircumferential direction.
  Position Pos_local(const int c, const int r, const int z);
  int Indexof(const int c, const int r, const int z);

public:

  Doppler();
  /// Constructor takes the flowfield and the colormap.
  Doppler(Flowfield& ff, Colormap& cmap);
  Doppler& operator=(const Doppler& cp);
  Doppler(const Doppler& cp);
  virtual ~Doppler(void);
  virtual void update(int /*framecount*/) = 0;

};

/// Doppler measuring the absolute speeds in the cone.
/**
  This tool does not represent any real endovascular probe and is here
  for academic reasons.
*/

class SpeedDoppler : public Doppler
{
protected:

  virtual const Float Valueof(const PosTimestep& pt);

public:

  SpeedDoppler();
```

```
  /// Constructor takes the flowfield and the colormap.
  SpeedDoppler(Flowfield& ff, Colormap& cm);

  SpeedDoppler& operator=(const SpeedDoppler& cp);

  SpeedDoppler(const SpeedDoppler& scp);

  virtual ~SpeedDoppler(void);

};

/// Doppler measuring the axial speeds in the cone.
/**
  This tool models the real Cardiometrics flowire behavior of axial speed
  sampling in a conical volume.  It writes minimum and maximum values to
  the log file xdoppler.dat every timestep.  The script xdoppler can be run
  to display an animated strip chart of the current axial speed range.
*/

class NormSpeedDoppler : public Doppler
{

private:

  ofstream* Logfile;
  int Logcount;

protected:

  virtual const Float Valueof(const PosTimestep& pt);

public:

  NormSpeedDoppler();

  /// Constructor takes the flowfield and the colormap.
  NormSpeedDoppler(Flowfield& ff, Colormap& cm);

  NormSpeedDoppler& operator=(const NormSpeedDoppler& cp);

  NormSpeedDoppler(const NormSpeedDoppler& scp);

  virtual ~NormSpeedDoppler(void);

  /// Update manages the virtual tool and outputs to the logfile.
  virtual void update(int frame_count);

};

#endif /* !__Doppler_h */
```

B.21 Doppler.cc

```
#include "Doppler.h"

Position Doppler::Pos_local(const int c, const int r, const int z)
{
  assert(c >= 0 && c < C_samples);
  assert(r >= 0 && r < R_samples);
  assert(z >= 0 && z < Z_samples);
  Float zval = (Float)z/(Float)(Z_samples - 1)*Height;
  Float radius_fraction = ((Float)r/(Float)(R_samples - 1));
  Float full_radius = zval*tan(Halfangle);
  Float real_radius = radius_fraction*full_radius;
  Float circum_angle = 2*PI*(Float)c/(Float)C_samples;
  Float xval = real_radius*cos(circum_angle);
  Float yval = real_radius*sin(circum_angle);
  return Position(xval, yval, zval);
};

int Doppler::Indexof(const int c, const int r, const int z)
{
  assert(c >= 0 && c < C_samples);
  assert(r >= 0 && r < R_samples);
  assert(z >= 0 && z < Z_samples);
  assert(   r == (R_samples - 1)
 || z == (Z_samples - 1)
 || z == 0);

  int base_start = 1 + C_samples*(Z_samples - 1);
  assert(base_start < Shape.numverts());

  if (z == 0) return 0;     // the apex
  if (r == (R_samples - 1)) // the sides up to the top
    {
      int retval = 1 + c + (z - 1)*C_samples;
      assert(retval >= 0 && retval < base_start);
      return retval;
    };
  if (r == 0 && z == (Z_samples - 1)) return base_start; // the base center

  if (r != 0 && r != (R_samples - 1) && z == (Z_samples - 1)) // base
    {
      int retval = base_start + 1 + c + (r - 1)*C_samples;
      assert(retval > base_start && retval < Shape.numverts());
      return retval;
    };

  assert(false); // should never reach here!!!
  return -1;
};

Doppler::Doppler()
{};
```

```
Doppler::Doppler(Flowfield& ff, Colormap& cmap)
: StatObjTool(ff,cmap,Position(0,0,-2.1),Position(0,0,-2.3)),
  Halfangle(29.0*PI/180.0), Height(4.2),
  Z_samples(6), R_samples(4), C_samples(24)
{
  int num_of_verts =   C_samples*(Z_samples - 1) + 1  // the outside
    + (R_samples - 2)*C_samples + 1; // the inner base
  assert(num_of_verts > 0);
  int thisindex = 0;

  // begin creating the object
  Shape.openobject(num_of_verts);

  // add the vertices
  // first the outside surface
  Shape.addvertex(Pos_local(0,0,0));  // the apex
  Shape.setvertexcolor(thisindex, Color(0,0,0));
  Shape.setvertexnormal(thisindex++, Position(0.0, 0.0, -1.0));

  for (int z = 1; z < Z_samples; z++)
    for (int c = 0; c < C_samples; c++)
      {
Shape.addvertex(Pos_local(c, R_samples - 1, z));
Float alpha = Halfangle;
Float beta = 2*PI*(Float)c/(Float)C_samples;
Shape.setvertexcolor(thisindex, Color(0,0,0));
Shape.setvertexnormal(thisindex++,
      Position(cos(alpha)*cos(beta),
       cos(alpha)*sin(beta),
       -sin(alpha)
       )
      );
      };

  // the base
  Shape.addvertex(Pos_local(0,0,Z_samples - 1)); // the center
  Shape.setvertexcolor(thisindex, Color(0,0,0));
  Shape.setvertexnormal(thisindex++, Position(0.0, 0.0, 1.0));
  for (int r = 1; r < R_samples-1; r++)
    for (int c = 0; c < C_samples; c++)
      {
Shape.addvertex(Pos_local(c, R_samples-1, Z_samples - 1));
Shape.setvertexcolor(thisindex, Color(0,0,0));
Shape.setvertexnormal(thisindex++, Position(0.0, 0.0, 1.0));
      };

  assert(thisindex == Shape.numverts()); // make sure we didn´t miscount

  // add the polygons

  // around the apex
  for (int c = 0; c < C_samples; c++)
    Shape.addpoly(Indexof(0,0,0),
  Indexof((c+1)%C_samples, R_samples-1, 1),
```

```
  Indexof(c, R_samples-1, 1)
  );

  // the rest of the outside face (except the base)
  for (int z = 1; z < Z_samples-1; z++)
    for (int c = 0; c < C_samples; c++)
      {
Shape.addpoly(Indexof(c, R_samples-1, z),
      Indexof((c + 1)%C_samples, R_samples-1, z),
      Indexof(c, R_samples-1, z + 1)
      );
Shape.addpoly(Indexof((c + 1)%C_samples, R_samples-1, z),
      Indexof((c + 1)%C_samples, R_samples-1, z + 1),
      Indexof(c, R_samples-1, z + 1)
      );
      };

  // around the center of the base
  for (int c = 0; c < C_samples; c++)
    Shape.addpoly(Indexof(0,0,Z_samples - 1),
  Indexof(c, 1, Z_samples - 1),
  Indexof((c+1)%C_samples, 1, Z_samples - 1)
  );

  // the rest of the base
  for (int r = 1; r < R_samples - 1 ; r++)
    for (int c = 0; c < C_samples; c++)
      {
Shape.addpoly(Indexof(c, r, Z_samples - 1),
      Indexof(c, r + 1, Z_samples - 1),
      Indexof((c+1)%C_samples, r + 1, Z_samples - 1)
      );
Shape.addpoly(Indexof(c, r, Z_samples - 1),
      Indexof((c+1)%C_samples, r + 1, Z_samples - 1),
      Indexof((c+1)%C_samples, r, Z_samples - 1)
      );
      };

  Shape.closeobject();
  Position pivpt(0,0,-2.1);
  Shape.setpivot(pivpt);
  FlowVisTool::setposition(Shape.getposition());
  FlowVisTool::setorientation(Shape.getorientation());
};

Doppler& Doppler::operator=(const Doppler& d)
{
  if (this == &d) return *this;
  StatObjTool::operator=(d);
  Halfangle = d.Halfangle;
  Height = d.Height;
  Z_samples = d.Z_samples;
  R_samples = d.R_samples;
  C_samples = d.C_samples;
```

```
};

Doppler::Doppler(const Doppler& d)
{
  *this = d;
};

Doppler::~Doppler(void)
{
};

const Float SpeedDoppler::Valueof(const PosTimestep& pt)
{
  Flowpoint fp = ff().eval(pt);
  Float spd = fp.speed();
  return spd;
};

SpeedDoppler::SpeedDoppler()
: Doppler()
{};

SpeedDoppler::SpeedDoppler(Flowfield& ff, Colormap& cm)
: Doppler(ff, cm)
{};

SpeedDoppler& SpeedDoppler::operator=(const SpeedDoppler& cp)
{
  if (this == &cp) return *this;
  Doppler::operator=(cp);
  return *this;
};

SpeedDoppler::SpeedDoppler(const SpeedDoppler& scp)
{
  *this = scp;
};

SpeedDoppler::~SpeedDoppler(void)
{ };

const Float NormSpeedDoppler::Valueof(const PosTimestep& pt)
{
  Flowpoint fp = ff().eval(pt);
  Quaternion q = orientation();
  Float spd = fp.speed(q);
  return spd;
};

NormSpeedDoppler::NormSpeedDoppler()
: Doppler()
{};

NormSpeedDoppler::NormSpeedDoppler(Flowfield& ff, Colormap& cm)
```

```
: Doppler(ff, cm)
{
  Logfile = new ofstream("xdoppler.dat");
  Logcount = 0;
};

NormSpeedDoppler& NormSpeedDoppler::operator=(const NormSpeedDoppler& cp)
{
  if (this == &cp) return *this;
  Doppler::operator=(cp);
  return *this;
};

NormSpeedDoppler::NormSpeedDoppler(const NormSpeedDoppler& scp)
{
  *this = scp;
};

NormSpeedDoppler::~NormSpeedDoppler(void)
{
  delete Logfile;
};

void NormSpeedDoppler::update(int frame_count)
{
  StatObjTool::update(frame_count);
  Range<Float> rng;
  for (int c = 0; c < C_samples; c++)
    for (int r = 0; r < R_samples; r++)
      for (int z = 0; z < Z_samples; z++)
{
  Position localpos = Pos_local(c,r,z);
  Position globalpos = position() + localpos*orientation();
  PosTimestep pt(globalpos, frame_count);
  rng.add(Valueof(pt));
};
  *Logfile << Logcount++ << " " << rng.min() << " "
    << rng.max() << "\n" << flush;
};
```

B.22 Skin.h

```
#ifndef __Skin_h
#define __Skin_h

#include "./wt.h"

#include "Flowfield.h"
#include "Colormap.h"
#include "FlowVisTool.h"
#include "utils.h"
#include "WTobj.h"
```

```
/// Colormapped value on surface of flow field.
class Skin : public FlowVisTool
{
protected:

  const int Numslicemax = 50;
  int Numslice;
  int Lastslice;
  WTobj* Skins[Numslicemax];
  virtual void Drawskin(void);
  virtual Float Valueof(int facenum, int vertnum, int slice) = 0;
  Colormap* Cm;

public:

  /// Constructor takes flow field and color map.
  Skin(Flowfield& ff, Colormap& cmap);

  virtual ~Skin();

  /// Update method to call every simulation loop.
  void update(int frame_count);

  /// Turn off the skin graphic.
  void deactivate(void);

};


class PlainSkin : public Skin
{
protected:

  virtual Float Valueof(int facenum, int vertnum, int slice);

public:

  /// Constructor takes flow field and color map.
  PlainSkin(Flowfield& ff, Colormap& cmap);

  virtual ~PlainSkin();
};



class WireFrame : public Skin
{
protected:

  virtual Float Valueof(int facenum, int vertnum, int slice);
  virtual void Drawskin(void);

public:
  /// Constructor takes flow field and color map.
```

```
  WireFrame(Flowfield& ff, Colormap& cmap);

  virtual ~WireFrame();
};


class PressureSkin : public Skin
{
protected:

  virtual Float Valueof(int facenum, int vertnum, int slice);

public:
  /// Constructor takes flow field and color map.
  PressureSkin(Flowfield& ff, Colormap& cmap);

  // should have assignment op and copy ctor here...

  virtual ~PressureSkin();

};

class StrainRateSkin : public Skin
{
protected:

  virtual Float Valueof(int facenum, int vertnum, int slice);

public:
  /// Constructor takes flow field and color map.
  StrainRateSkin(Flowfield& ff, Colormap& cmap);

  // should have assignment op and copy ctor here...

  virtual ~StrainRateSkin();
};


#endif /* !__Skin_h */
```

B.23 Skin.cc

```
#include "Skin.h"

void Skin::Drawskin(void)
{
  Numslice = ff().num_time();
  if (Numslice > Numslicemax)
    {
      cerr << "Error in Skin construction: Number of time slices > max.\n";
      cerr << " setting equal to max.\n";
      Numslice = Numslicemax;
```

```
    };

  Lastslice = 0;

  for (int slice = 0; slice < Numslice; slice++)
    {
      int numverts = ff().num_exterior()*3;
      Skins[slice] = new WTobj();
      Skins[slice]->openobject(numverts);
      for (int facenum = 0; facenum < ff().num_exterior(); facenum++)
{
  for (int vertnum = 0; vertnum < 3; vertnum++)
    {
      Position nodepos
= ff().node(ff().exterior(facenum).ni(vertnum)).pos();
      Skins[slice]->addvertex(nodepos);
      Float val = Valueof(facenum,vertnum,slice);
      Color c = Cm->color(val);
      Skins[slice]->setvertexcolor(facenum*3 + vertnum, c);
      Position norm = ff().exterior(facenum).normal();
      Skins[slice]->setvertexnormal(facenum*3 + vertnum, norm);
    };
  Skins[slice]->addpoly(facenum*3, facenum*3 + 1, facenum*3 + 2);
};
      Skins[slice]->blend();
      Skins[slice]->closeobject();
      Skins[slice]->optimize();
    };
};

Skin::Skin(Flowfield& ff,  Colormap& cmap)
: FlowVisTool(ff)
{
  Cm = &cmap;
};

Skin::~Skin()
{
  for (int i = 0; i < Numslice; i++)
    {
      delete Skins[i];
    };
};

void Skin::update(int frame_count)
{
  if (active())
    {
      Skins[Lastslice]->setvisibility(false);
      Skins[frame_count]->setvisibility(true);
      Lastslice = frame_count;
    };
};
```

```
void Skin::deactivate(void)
{
  FlowVisTool::deactivate();
  Skins[Lastslice]->setvisibility(false);
};

void WireFrame::Drawskin(void)
{
  Numslice = ff().num_time();
  if (Numslice > Numslicemax)
    {
      cerr << "Error in Skin construction: Number of time slices > max.\n";
      cerr << " setting equal to max.\n";
      Numslice = Numslicemax;
    };

  Lastslice = 0;

  for (int slice = 0; slice < Numslice; slice++)
    {
      int numverts = ff().num_exterior()*3*2;
      Skins[slice] = new WTobj();
      Skins[slice]->openobject(numverts);
      for (int facenum = 0; facenum < ff().num_exterior(); facenum++)
{
  for (int vertnum = 0; vertnum < 3; vertnum++)
    {
      Position nodepos
= ff().node(ff().exterior(facenum).ni(vertnum)).pos();
      Position nodepos2 = nodepos
+ Position(0.025 * ((facenum * vertnum )%2),
   0.025 * ((facenum * vertnum + 1)%2),
   0.025 * ((facenum * vertnum )%2));
      Skins[slice]->addvertex(nodepos);
      Skins[slice]->addvertex(nodepos2);
      Float val = Valueof(facenum,vertnum,slice);
      Color c = Cm->color(val);
      Skins[slice]->setvertexcolor((facenum*3 + vertnum)*2, c);
      Skins[slice]->setvertexcolor((facenum*3 + vertnum)*2 + 1, c);
      Position norm = ff().exterior(facenum).normal();
      Skins[slice]->setvertexnormal((facenum*3 + vertnum)*2, norm);
      Skins[slice]->setvertexnormal((facenum*3 + vertnum)*2 + 1, norm);
    };
  int base = facenum*3*2;
  Skins[slice]->addpoly(base    , base + 1, base + 2);
  Skins[slice]->addpoly(base + 2, base + 3, base + 4);
  Skins[slice]->addpoly(base + 4, base + 5, base + 0);
};
      Skins[slice]->closeobject();
      Skins[slice]->optimize();
    };
};
```

```
Float WireFrame::Valueof(int facenum, int vertnum, int slice)
{
  return 0.5;
};

WireFrame::WireFrame(Flowfield& ff, Colormap& cmap)
: Skin(ff,cmap)
{
  Drawskin();
};

WireFrame::~WireFrame()
{};

Float PlainSkin::Valueof(int facenum, int vertnum, int slice)
{
  return 0.5;
};

PlainSkin::PlainSkin(Flowfield& ff, Colormap& cmap)
: Skin(ff,cmap)
{
  Drawskin();
};

PlainSkin::~PlainSkin()
{};

Float PressureSkin::Valueof(int facenum, int vertnum, int slice)
{
  return ff().node(ff().exterior(facenum).ni(vertnum)).fp(slice).pressure();
};

PressureSkin::PressureSkin(Flowfield& ff, Colormap& cmap)
: Skin(ff,cmap)
{
  Drawskin();
};

PressureSkin::~PressureSkin()
{};

Float StrainRateSkin::Valueof(int facenum, int vertnum, int slice)
{
  Flowpoint fpt = ff().node(ff().exterior(facenum).ni(vertnum)).fp(slice);
  Linearfield lf = ff().cell((ff().exterior(facenum).this_cell()))
                          .linfld(slice);
  Position norm = ff().exterior(facenum).normal();
  StrainRate sr = lf.strainrate();
  Position surfsr = sr.surfstrainrate(norm);
  Float val = surfsr.magnitude();
  return val;
};
```

```
StrainRateSkin::StrainRateSkin(Flowfield& ff, Colormap& cmap)
: Skin(ff,cmap)
{
  Drawskin();
};

StrainRateSkin::~StrainRateSkin()
{};
```

B.24 va.cc

```
#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include <stdlib.h>

extern "C" {
#include "./wt.h"
};

#include "Flowfield.h"
#include "Colormap.h"
#include "utils.h"
#include "DynVar.h"
#include "VirtualObject.h"
#include "Hand.h"
#include "Skin.h"
#include "Widget.h"
#include "FlowVisTool.h"
#include "Cuttingplane.h"
#include "Doppler.h"
#include "sys/time.h"

//
// Global declaration of all WorldToolKit Objects
//

WTsensor* mouse;           // 2D Mouse
WTsensor* fmouse;          // flying mouse
WTp3 deltap;               // the amount to move on every step.
WTp3 deltao;               // the amount to rotate (in radians) about the axes

// list of tool pointers
list<FlowVisTool*> vtoollist;

// list of widgets pointers
list<Widget*> widglist;

// hand pointer
Hand* hand;

// path variables...
const int numvpmarks = 10;
```

```
WTpathnode* beginpnode[numvpmarks];
WTpathnode* endpnode[numvpmarks];
WTpath* defpath[numvpmarks];
WTpath* defpath2[numvpmarks];
WTobject* pathmarker;
WTp3 endpnodepos[numvpmarks];
WTq endpnodeori[numvpmarks];

int frame_count = 0; bool timestop = false; int frame_count_num; const
hrtime_t timebetweensimupdates = 100000000; // in nanoseconds
hrtime_t lastupdate;

Flowfield Ff;

// I want to move objects before I do my actions, otherwise the objects appear
// to get ahead of themselves.
void setup_eventorder(void)
{
  const short nevents = 4;
  short events[nevents];
  events[0] = WTEVENT_OBJECTSENSOR;
  events[1] = WTEVENT_ACTIONS;
  events[2] = WTEVENT_TASKS;
  events[3] = WTEVENT_PATHS;
  WTuniverse_seteventorder(nevents, events);
};

void setup_viewpoint(void)
{
  WTpq save;

  // get the position and orientation of the viewpoint
  WTviewpoint_getposition(WTuniverse_getviewpoint(), save.p);
  WTviewpoint_getorientation(WTuniverse_getviewpoint(), save.q);

  // zoom until roughly all of universe is in view
  WTviewpoint_moveto(WTuniverse_getviewpoint(), &save);
  WTviewpoint_zoomall(WTuniverse_getviewpoint());

  // these are hacked settings which sometimes are good...
  save.p[0] = 8.55;
  save.p[1] = 1.43;
  save.p[2] = -15.3;
  save.q[0] = 0.204215;
  save.q[1] = 0.0275671;
  save.q[2] = -0.978237;
  save.q[3] = 0.0242637;

  WTviewpoint_setposition(WTuniverse_getviewpoint(), save.p);
  WTviewpoint_setorientation(WTuniverse_getviewpoint(), save.q);

  // scale parallax
  WTviewpoint_setparallax(WTuniverse_getviewpoint(), 0.3);
};
```

```
void setup_sensors(void)
{
  // try to initialize a Mouse sensor
  mouse = WTsensor_new(WTmouse_open, WTmouse_close,
WTmouse_moveview2, NULL, 1,
WTSENSOR_DEFAULT);
  if (!mouse) WTerror("Couldn´t find mouse\n");

  // try to initialize a flying mouse sensor
  fmouse = 0;
  while (!fmouse)
    {
      fmouse = WTbaron_new(SERIAL1);
      if (!fmouse)
cerr << "Couldn´t find flying mouse\nPlease reposition";
      system("sleep 5");
    };

  // initialize sensitivities
  WTsensor_setsensitivity(fmouse, 1.0);

  // attach user viewpoint to mouse
  WTviewpoint_addsensor(WTuniverse_getviewpoint(), mouse);

};

void setup_lights(void)
{
  WTlight_load("lights");
};

void display_info(void)
{
  // frame rate and total polygons
  cerr << "Polygons: " << WTuniverse_npolygons()
       << "  Frame rate: " << WTuniverse_framerate() << " fps\n";
  WTpq save;
  // get the position and orientation of the viewpoint
  WTviewpoint_getposition(WTuniverse_getviewpoint(), save.p);
  WTviewpoint_getorientation(WTuniverse_getviewpoint(), save.q);
  cerr << "Position: " << save.p[0] << " " << save.p[1]
       << " " << save.p[2] << "\n";
  cerr << "Orientation: " << save.q[0] << " " << save.q[1]
       << " " << save.q[2] << " " << save.q[3] << "\n";
  cerr.flush();
};


void initvpmarks(void)
{
  pathmarker = WTobject_newblock(0.2, 0.2, 0.2, FALSE, FALSE);
  WTobject_setvisibility(pathmarker, FALSE);
  for (int i = 0; i < numvpmarks; i++)
```

```
    {
      defpath[i] = 0;
      defpath2[i] = 0;
    };

  // hack here to set up base0 vps 3 and 4 (delta/eps) to be AP and
  // lateral views from swine 895
     endpnodepos[3][0] = -0.110125;
     endpnodepos[3][1] = 23.1827;
     endpnodepos[3][2] = 13.2137;

     endpnodeori[3][0] = -0.443775;
     endpnodeori[3][1] = -0.5204;
     endpnodeori[3][2] = 0.542649;
     endpnodeori[3][3] = 0.487626;

     endpnodepos[4][0] = 22.9;
     endpnodepos[4][1] = 1.65108;
     endpnodepos[4][2] = 11.6422;

     endpnodeori[4][0] = -0.658208;
     endpnodeori[4][1] = 0.0072391;
     endpnodeori[4][2] = 0.751819;
     endpnodeori[4][3] = 0.0384458;
};

void setvpmark(const int i)
{
  WTviewpoint_getposition(WTuniverse_getviewpoint(), endpnodepos[i]);
  WTviewpoint_getorientation(WTuniverse_getviewpoint(), endpnodeori[i]);
};

void gotovpmark(const int i)
{

  if (defpath[i]) WTpath_delete(defpath[i]);
  if (defpath2[i]) WTpath_delete(defpath2[i]);
  beginpnode[i] = WTpathnode_new(0);
  endpnode[i] = WTpathnode_new(0);
  WTpathnode_setposition(endpnode[i], endpnodepos[i]);
  WTpathnode_setorientation(endpnode[i], endpnodeori[i]);
  WTp3 beginpnodepos;
  WTq beginpnodeori;
  WTviewpoint_getposition(WTuniverse_getviewpoint(), beginpnodepos);
  WTviewpoint_getorientation(WTuniverse_getviewpoint(), beginpnodeori);
  WTpathnode_setposition(beginpnode[i], beginpnodepos);
  WTpathnode_setorientation(beginpnode[i], beginpnodeori);
  defpath[i] = WTpath_new(pathmarker);
  WTpath_setvisibility(defpath[i], FALSE);
  WTpath_appendnode(defpath[i], beginpnode[i]);
  WTpath_appendnode(defpath[i], endpnode[i]);
  defpath2[i] = WTpath_interpolate(defpath[i], 100, WTPATH_LINEAR);
  WTpath_setvisibility(defpath2[i], FALSE);
  WTpath_play(defpath2[i]);
```

```
};


/*
 * Universe action function.
 *
 */

static void actionfn(void)
{
  if (!timestop)
    {
      if ((gethrtime() - lastupdate) > timebetweensimupdates)
{
  lastupdate = gethrtime();
  frame_count++;
  if (frame_count == frame_count_num) frame_count = 0;
};
    };

  // update hand
  hand->update(frame_count);

  bool leftbutton = (WTsensor_getmiscdata(fmouse) & WTLOGITECH_LEFTBUTTON);

  if (!leftbutton && hand->isgrabbing())         // release widgets
    {
      hand->ungrab();
    }
  else if (leftbutton && !(hand->isgrabbing()))  // try to grab a widget
    {
      list<Widget*>::iterator whichwidg = widglist.begin();
      bool foundit = false;
      while (!foundit && (whichwidg != widglist.end()))
{
  if (hand->cangrab(*whichwidg))
    {
      hand->grab(*whichwidg);
      foundit = true;
    };
  whichwidg++;
};
    };

  // update grabbed widget
  if (hand->isgrabbing())
    {
      hand->grabbed()->update(frame_count);
    };

  // update any tools which are on
  {  // update ungrabbed widgets
    list<FlowVisTool*>::iterator whichtool = vtoollist.begin();
    while (whichtool != vtoollist.end())
```

```
      {
if ((*whichtool)->active())
  {
    (*whichtool)->update(frame_count);
  };
whichtool++;
      };
  };

  {  // update ungrabbed widgets
    list<Widget*>::iterator whichwidg = widglist.begin();
    while (whichwidg != widglist.end())
      {
if (!(*whichwidg)->grabbed())
  {
    (*whichwidg)->update(frame_count);
    if (hand->cangrab(*whichwidg))
      {
(*whichwidg)->hilite();
      }
    else
      {
(*whichwidg)->lolite();
      };
  };
whichwidg++;
      };
  };

  // update viewpoint
  WTviewpoint_translate(WTuniverse_getviewpoint(), deltap, WTFRAME_LOCAL);
  for (int i = 0; i < 3; i++)
    {
      WTviewpoint_rotate(WTuniverse_getviewpoint(), i,
 deltao[i], WTFRAME_LOCAL);
    };

  unsigned int key;
  key = WTkeyboard_getlastkey();  // get key presses, if any
  // handle keyboard
  if (key == ´q´) {// quit simulation
    WTuniverse_stop();
  } else if (key == ´z´) {// information
    display_info();
  } else if (key >= ´1´ && key <= (´0´+vtoollist.size())) { // toggle tool
    int toolnum = key - ´1´;
    list<FlowVisTool*>::iterator whichtool = vtoollist.begin();
    for (int i = 0; i < toolnum; i++) whichtool++;
    if ((*whichtool)->active())
      { (*whichtool)->deactivate(); }
    else
      { (*whichtool)->activate(); };
  } else if (key == ´{´) {// decrease parallax
    WTviewpoint_setparallax(WTuniverse_getviewpoint(),
```

```
    WTviewpoint_getparallax(WTuniverse_getviewpoint())
* 1.0/1.2);
    cerr << "Parallax : "
         << WTviewpoint_getparallax(WTuniverse_getviewpoint())
         << "\n";
  } else if (key == ´}´) {// increase parallax
    WTviewpoint_setparallax(WTuniverse_getviewpoint(),
    WTviewpoint_getparallax(WTuniverse_getviewpoint())
* 1.2);
    cerr << "Parallax : "
         << WTviewpoint_getparallax(WTuniverse_getviewpoint())
         << "\n";
  } else if (key == ´c´) {// center hand
    hand->centerinvp(WTuniverse_getviewpoint());
  } else if (key == ´j´) {// stop all movement
    deltap[0] = deltap[1] = deltap[2] = 0;
    deltao[0] = deltao[1] = deltao[2] = 0;
  } else if (key == ´u´) {// move forward
    deltap[2] += 0.1;
  } else if (key == ´h´) {// move left
    deltap[0] -= 0.1;
  } else if (key == ´k´) {// move right
    deltap[0] += 0.1;
  } else if (key == ´m´) {// move back
    deltap[2] -= 0.1;
  } else if (key == ´y´) {// move up
    deltap[1] -= 0.1;
  } else if (key == ´n´) {// move down
    deltap[1] += 0.1;
  } else if (key == ´H´) {// rotate left
    deltao[1] -= 0.01;
  } else if (key == ´K´) {// rotate right
    deltao[1] += 0.01;
  } else if (key == ´Y´) {// rotate up
    deltao[0] += 0.01;
  } else if (key == ´N´) {// rotate down
    deltao[0] -= 0.01;
  } else if (key == ´o´) {// roll left
    deltao[2] -= 0.01;
  } else if (key == ´p´) {// roll right
    deltao[2] += 0.01;
  } else if (key == ´A´) {// set end path node
    setvpmark(0);
  } else if (key == ´a´) {// set goto end path node
    gotovpmark(0);
  } else if (key == ´B´) {// set end path node
    setvpmark(1);
  } else if (key == ´b´) {// set goto end path node
    gotovpmark(1);
  } else if (key == ´G´) {// set end path node
    setvpmark(2);
  } else if (key == ´g´) {// set goto end path node
    gotovpmark(2);
  } else if (key == ´D´) {// set end path node
```

```
    setvpmark(3);
  } else if (key == ´d´) {// set goto end path node
    gotovpmark(3);
  } else if (key == ´E´) {// set end path node
    setvpmark(4);
  } else if (key == ´e´) {// set goto end path node
    gotovpmark(4);
  } else if (key == ´t´) {// toggle timestop
    if (timestop) { timestop = false; } else { timestop = true; };
  } else if (key == ´+´) {// inc timestep
    if (timestop) {
      frame_count++;
      if (frame_count == frame_count_num) frame_count = 0;
    };
  } else if (key == ´-´) {// dec timestep
    if (timestop) {
      frame_count--;
      if (frame_count == -1) frame_count = frame_count_num - 1;
    };
  };
};


int
main()
{
  ifstream ff_file("standard.ff");
  ff_file >> Ff;
  Ff.status();
  frame_count_num = Ff.num_time();

  const long display_conf = WTDISPLAY_CRYSTALEYES;
//  const long display_conf = WTDISPLAY_DEFAULT;
  WTuniverse_new(display_conf, WTWINDOW_DEFAULT); // init the universe
  WTuniverse_setbgcolor(0x111);

  setup_eventorder();
  setup_sensors();
  setup_lights();
  setup_viewpoint();

  Range<Float> plainrange(0.0,1.0);
  ColormapBlackGrayWhite plaincmap(plainrange); // for plain geometry

  Range<Float> pressrange(Ff.flowpointrange().min().pressure(),
  Ff.flowpointrange().max().pressure());
  ColormapBlueGreenRed presscmap(pressrange);   // pressure
  // surface strain rate color map
  Ff.surfstrainraterange().status();
  ColormapBlackRedYellow surfstrainratecmap(Ff.surfstrainraterange());
  ColormapBlackRedYellow absstrainratecmap(Ff.absstrainraterange());
  Float maxspeed = Ff.speedrange().max();
  Float minspeed = Ff.speedrange().min();
  Float absspeed = ((maxspeed > -minspeed) ? maxspeed : -minspeed);
```

```
  ColormapBlackRedYellow speedcmap(Ff.speedrange());   // absolute speed

  Range<Float> posnegspeedrange(-absspeed, absspeed);  // pos/neg speed
  ColormapCyanBlueBlackRedYellow posnegspeedcmap(posnegspeedrange);

  // make the hand.
  hand = new Hand(fmouse);

  // make the tools and tool pointer list.
  WireFrame* tmpwireframeptr = new WireFrame(Ff, plaincmap);
  tmpwireframeptr->activate();
  vtoollist.push_back(tmpwireframeptr);
  vtoollist.push_back(new PressureSkin(Ff, presscmap));
  vtoollist.push_back(new StrainRateSkin(Ff, surfstrainratecmap));
  vtoollist.push_back(new SpeedCuttingplane(Ff, speedcmap));
  vtoollist.push_back(new NormSpeedCuttingplane(Ff, posnegspeedcmap));
  vtoollist.push_back(new AbsStrainRateCuttingplane(Ff, absstrainratecmap));
  // this tool should really have its own cmap...
  vtoollist.push_back(new NormSpeedDoppler(Ff, posnegspeedcmap));
  vtoollist.push_back(new PlainSkin(Ff, plaincmap));

  // make the widget pointer list.
  list<FlowVisTool*>::iterator whichvt = vtoollist.begin();
  while (whichvt != vtoollist.end())
    {
      list<Widget*>::iterator whichwidg = (*whichvt)->widgetlist().begin();
      while (whichwidg != (*whichvt)->widgetlist().end())
widglist.push_back(*whichwidg++);
      whichvt++;
    };

  initvpmarks();

  lastupdate = gethrtime();
  WTuniverse_setactions(actionfn);// Set action function for button presses
  WTkeyboard_open();   // open the keyboard

  WTuniverse_ready();  // Prepare the universe for start of the simulation.

  WTuniverse_go();     // enter the main simulation loop
  WTuniverse_delete(); // clean everything up
};
```

Bibliography

[AJ87] James G. Anderson and Stephen J. Jay. The diffusion of computer applications in medical settings. In James G. Anderson and Stephen J. Jay, editors, *Use and Impact of Computers in Clinical Medicine*, pages 2–14. Springer-Verlag, 1987.

[And84] John D. Anderson Jr. *Fundamentals of Aerodynamics*. McGraw-Hill, Inc., 1984.

[BC94] Grigore Burdea and Philippe Coiffet. *Virtual Reality Technology*. John Wiley & Sons, Inc., 1994.

[BDH+94] Manfred Brill, Wladimir Djatschin, Hans Hagen, Stanislav V. Klimenko, and Hans-Christian Rodrian. Streamball techniques for flow visualization. In R. Daniel Bergeron and Arie E. Kaufman, editors, *IEEE Visualization '94*, pages 225–231, Los Alamitos, California, October 1994. IEEE, IEEE Computer Society Press.

[BF92a] Clifford Beshers and Steven Feiner. Automated design of virtual worlds for visualizing multivariate relations. In Arie E. Kaufman and Gregory M. Nielson, editors, *IEEE Visualization '92*, pages 283–290, Los Alamitos, California, October 1992. IEEE, IEEE Computer Society Press.

[BF92b] Steve Bryson and Scott S. Fisher. Defining, modeling, and measuring system lag in virtual environments. In *SIGGRAPH Course 9: Implementation of Immersive Virtual Environments*, pages 9.1–9.12. ACM, 1992.

[BL91] Steve Bryson and Creon Levit. The virtual windtunnel: An environment for the exploration of three-dimensional unsteady flows. In Gregory M. Nielson and Larry Rosenblum, editors, *IEEE Visualization '91*, pages 17–24, Los Alamitos, California, October 1991. IEEE, IEEE Computer Society Press.

[BL92] S. Bryson and C. Levit. The virtual wind tunnel. *IEEE Computer Graphics and Applications.*, 12:25–34, July 1992.

[BM90] P. K. Banerjee and L. Morino, editors. *Boundary Element Methods in Nonlinear Fluid Dynamics*. Elsevier Applied Science, 1990.

[Boh87] G. Bohme. *Non-Newtonian Fluid Mechanics*. North Holland Publishing Company, 1987.

[Bro67] Robert S. Brodkey. *The Phenomena Of Fluid Motions*. Addison-Wesley Publishing Company, 1967.

[BSCWW93] Mikael Bostrom, Sunil K. Singh, and M.D. Christopher W. Wiley. Design of an interactive lumbar puncture simulator with tactile feedback. In *IEEE Virtual Reality Annual International Symposium*, pages 280–286, 1993.

[BST95] Armelle C. Burleson, Charles M. Strother, and Vincent T. Turitto. Computer modeling of intracranial saccular and lateral aneurysms for the study of their hemodynamics. *Neurosurgery*, 37(4):774–784, October 1995.

[Cab93] Brian Cabral. Imaging vector fields using line integral convolution. In *Computer Graphics Proceedings*, pages 263–270. ACM SIGGRAPH, ACM Press, August 1993.

[CMB94] Roger Crawfis, Nelson Max, and Barry Becker. Vector field visualization. *IEEE Computer Graphics and Applications*, 14(5):50–56, September 1994.

[CSH+92] D. Brookshire Conner, Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, Robert C. Zeleznik, and Andries van Dam. Three-dimensional widgets. In *1992 Symposium on Interactive 3D Graphics*, pages 183–188. ACM SIGGRAPH, ACM Press, March 1992.

[DG94] G. O. Domik and B. Gutkauf. User modeling for adaptive visualization systems. In R. Daniel Bergeron and Arie E. Kaufman, editors, *IEEE Visualization '94*, pages 217–223, Los Alamitos, California, October 1994. IEEE, IEEE Computer Society Press.

[DH92] Thierry Delmarcelle and Lambertus Hesselink. Visualization of second order tensor fields and matrix data. In Arie E. Kaufman and Gregory M. Nielson, editors, *IEEE Visualization '92*, pages 316–323, Los Alamitos, California, October 1992. IEEE, IEEE Computer Society Press.

[Din85] Leopold Dintenfass. *Blood Viscosity, Hyperviscosity & Hyperviscosaemia*. MTP Press Limited, 1985.

[dLvW93] Willem C. de Leeuw and Jarke J. van Wijk. A probe for local flow field visualization. In Gregory M. Nielson and Dan Bergeron, editors, *IEEE Visualization '93*, pages 39–45, Los Alamitos, California, October 1993. IEEE, IEEE Computer Society Press.

[EJ94] Rae A. Earnshaw and Mikael Jern. Fundamental approaches to interactive real-time visualization systems. In L. Rosenblum, R. A. Earnshaw, J. Encaracao, H. Hagen, A. Kaufman, S. Klimenko, G. Nielson, F. Post, and D. Thalmann, editors, *Scientific Visualization*, chapter 14, pages 223–238. Academic Press, 1994.

[FH94] Jean M. Favre and James Hahn. An object oriented design for the visualization of multi-variable data objects. In R. Daniel Bergeron and Arie E. Kaufman, editors, *IEEE Visualization '94*, pages 318–325, Los Alamitos, California, October 1994. IEEE, IEEE Computer Society Press.

[Flua] Fluent Inc., Centerra Resource Park, 10 Cavendish Court, Lebanon, NH 03766. *Fluent Unstructured User's Manual.*

[Flub] Fluent Inc., Centerra Resource Park, 10 Cavendish Court, Lebanon, NH 03766. *NEKTON Version 2.85 Theory.*

[Fluc] Fluent Inc., Centerra Resource Park, 10 Cavendish Court, Lebanon, NH 03766. *NEKTON Version 2.85 User's Guide.*

[For94] Lisa K. Forssell. Visualizing flow over curvilinear grid surfaces using line integral convolution. In R. Daniel Bergeron and Arie E. Kaufman, editors, *IEEE Visualization '94*, pages 240–247, Los Alamitos, California, October 1994. IEEE, IEEE Computer Society Press.

[Fox83] John L. Fox. *Intracranial Aneurysms*, volume 1. Springer-Verlag, 1983.

[Fuc90] H. Fuchs. Systems for display of 3 dimensional medical data. In K.H. Hohne, H. Fuchs, and S.M. Pizer, editors, *3D Imaging in Medicine: Algorithms, Systems, Applications*, volume 60, pages 315–332. Springer Verlag, New York, NY, 1990. NATO Advanced Research Workshop on 3D Imaging in Medicine. Nato Advanced Science Institutes Series, Series F, Computer and Systems Science.

[GBT93] Enrico Gobbetti, Jean-Francis Balaguer, and Daniel Thalmann. Vb2 an architecture for interaction in synthetic worlds'. In *UIST 1993*, pages 167–178. ACM, November 1993.

[GCOM92] Carlos F. Gonzalez, Young I. Cho, Hector V. Ortega, and Jacgues Moret. Intracranial aneurysms: Flow analysis of their origin and progression. *AJNR*, 13:181–188, February 1992.

[GSPR92] Virgil B. Graves, Charles M. Strother, Curtis R. Partington, and Alan Rappe. Flow dynamics of lateral carotid artery aneurysms and their effects on coils and balloons: An experimental study in dogs. *AJNR*, 13:189–196, February 1992.

[GVDD91] Guido Guglielmi, Fernando Viñuela, Jacques Dion, and Gary Duckwiler. Electrothrombosis of saccular aneurysms via endovascular approach part 2: Preliminary clinical experience. *J. Neurosurg.*, 75:8–14, July 1991.

[GVSM91] Guido Guglielmi, Fernando Viñuela, Ivan Sepetka, and Velio Macellari. Electrothrombosis of saccular aneurysms via endovascular approach part 1: Electrochemical basis, technique, and experimental results. *J. Neurosurg.*, 75:1–7, July 1991.

[Har93] Michael R. Harreld. Computer simulation of blood flow in an intracranial aneurysm. Master's thesis, UCLA, 1993.

[HH90] J. L. Helman and Lambertus Hesselink. Surface representations of two- and three-dimensional fluid flow topology. In Arie Kaufman, editor, *IEEE Visualization '90*, pages 6–13, Los Alamitos, California, October 1990. IEEE, IEEE Computer Society Press.

[HM94] Andrew J. Hanson and Hui Ma. Visualizing flow with quaternion frames. In R. Daniel Bergeron and Arie E. Kaufman, editors, *IEEE Visualization '94*, pages 108–115, Los Alamitos, California, October 1994. IEEE, IEEE Computer Society Press.

[Kap87] Bonnie Kaplan. The influence of medical values and practices on medical computer applications. In James G. Anderson and Stephen J. Jay, editors, *Use and Impact of Computers in Clinical Medicine*, pages 39–50. Springer-Verlag, 1987.

[KG90] Olaf Kübler and Guido Gerig. Segmentation and analysis of multidimensional data-sets in medicine. In K.H. Hohne, H. Fuchs, and S.M. Pizer, editors, *3D Imaging in Medicine: Algorithms, Systems, Applications*, volume 60, pages 63–81. Springer Verlag, New York, NY, 1990. NATO Advanced Research Workshop on 3D Imaging in Medicine. Nato Advanced Science Institutes Series, Series F, Computer and Systems Science.

[KH91] R. Victor Klassen and Steven J. Harrington. Shadowed hedgehogs: A technique for visualizing 2D slices of 3D vector fields. In Gregory M. Nielson and Larry Rosenblum, editors, *IEEE Visualization '91*, pages 148–153, Los Alamitos, California, October 1991. IEEE, IEEE Computer Society Press.

[KN91] K. Koyamada and T. Nishio. Volume visualization of 3D finite element method results. *IBM Journal of Research and Development*, 35:12–25, January 1991.

[KN95] B. V. Rathish Kumar and K. B. Naidu. Finite element analysis of nonlinear pulsatile suspension flow dynamics in blood vessels with aneurysm. *Computers in Biology and Medicine*, 25(1):1–19, January 1995.

[LAC+92] Bruce Lucas, Gregory D. Abram, Nancy S. Collins, David A. Epstein, Donna L. Gresh, and Kevin P. McAuliffe. An architecture for a scientific visualization system. In Arie E. Kaufman and Gregory M. Nielson, editors, *IEEE Visualization '92*, pages 107–114, Los Alamitos, California, October 1992. IEEE, IEEE Computer Society Press.

[LB92] Creon Levit and Steve Bryson. Lessons learned while implementing the virtual windtunnel project. In *SIGGRAPH Course 9: Implementation of Immersive Virtual Environments*, pages 4.1–4.7. ACM, 1992.

[LC90] Stéphane Lavallée and Philippe Cinquin. Computer assisted medical interventions. In K.H. Hohne, H. Fuchs, and S.M. Pizer, editors, *3D Imaging in Medicine: Algorithms, Systems, Applications*, volume 60, pages 301–312. Springer Verlag, New York, NY, 1990. NATO Advanced Research Workshop on 3D Imaging in Medicine. Nato Advanced Science Institutes Series, Series F, Computer and Systems Science.

[LPR93] M. Löw, K. Perktold, and R. Raunig. Hemodynamics in rigid and distensible saccular aneurysms: a numerical study of pulsatile flow characteristics. *Biorheology*, 30(3 & 4):287–298, May 1993.

[LSPR87] D. W. Liepsch, H. J. Steiger, A. Poll, and H. J. Reulen. Hemodynamic stress in lateral saccular aneurysms. *Biorheology*, 24:689–710, 1987.

[MCG94] Nelson Max, Roger Crawfis, and Charles Grant. Visualizing 3D velocity fields near contour surfaces. In R. Daniel Bergeron and Arie E. Kaufman, editors, *IEEE Visualization '94*, pages 248–255, Los Alamitos, California, October 1994. IEEE, IEEE Computer Society Press.

[Mer93] J.R. Merril. Surgery on the cutting-edge: Virtual reality applications in medical education. *Virtual Reality World*, 1(3):34–38, November 1993.

[Mil89] William R. Milnor. *Hemodynamics*. Williams & Wilkins, 1989.

[MS92] Kwan-Liu Ma and Philip J. Smith. Virtual smoke: An interactive 3D flow visualization technique. In Arie E. Kaufman and Gregory M. Nielson, editors, *IEEE Visualization '92*, pages 46–53, Los Alamitos, California, October 1992. IEEE, IEEE Computer Society Press.

[NKS84] Hideyuki Niimi, Yosuke Kawano, and Ikuo Sugiyama. Structure of blood flow through a curved vessel with an aneurysm. *Biorheology*, 21:603–615, 1984.

[Osb80] Anne G. Osborn. *Introduction to Cerebral Angiography*. Harper & Row, 1980.

[PA94] Alex Pang and Naim Alper. Mix&match: A construction kit for visualization. In R. Daniel Bergeron and Arie E. Kaufman, editors, *IEEE Visualization '94*, pages 302–309, Los Alamitos, California, October 1994. IEEE, IEEE Computer Society Press.

[Pat80] Suhas V. Patankar. *Numerical heat transfer and fluid flow*. Hemisphere Publishing Corp., 1980.

[PKH+88] K. Perktold, T. Kenner, D. Hilbert, B. Spork, and H. Florian. Numerical blood flow analysis: arterial bifurcation with a saccular aneurysm. *Basic Res Cardiol*, 83:24–31, 1988.

[PLZ79] H. W. Pia, C. Langmaid, and J. Zierski, editors. *Cerebral Aneurysms*. Springer-Verlag, 1979.

[RC85] John A. Roberson and Clayton T. Crowe. *Engineering Fluid Mechanics*. Houghton Mifflin Company, 3rd edition, 1985.

[Red84] J. N. Reddy. *An Introduction To The Finite Element Method*. McGraw-Hill Book Company, 1984.

[Rhe91] Howard Rheingold. *Virtual Reality*. Summit Books, 1991.

[Rob90] Philip K. Robertson. A methodology for scientific data visualization: Choosing representations based on a natural scene paradigm. In Arie Kaufman, editor, *IEEE Visualization '90*, pages 114–123, Los Alamitos, California, October 1990. IEEE, IEEE Computer Society Press.

[RPM95] David Redmond-Pyle and Alan Moore. *Graphical User Interface Design and Evaluation*. Prentice Hall, 1995.

[RSS+91] J. Rosenman, S.L. Sailer, G.W. Sherouse, E.L. Chaney, and J.E. Tepper. Virtual simulation: Initial clinical results. *Int. J. Radiation Oncology Biol. Phys.*, 20:843–851, 1991.

[RT93] Bernice E. Rogowitz and Lloyd A. Treinish. An architecture for rule-based visualization. In Gregory M. Nielson and Dan Bergeron, editors, *IEEE Visualization '93*, pages 236–243, Los Alamitos, California, October 1993. IEEE, IEEE Computer Society Press.

[Sat93] R.M. Satava. On the cutting edge: VR and surgery. virtual reality '93: Fall special report. *AI Expert*, pages 1–44, September 1993.

[SBR+90] G.W. Sherouse, J.D. Bourland, K. Reynolds, H.L. McMurray, T.P. Mitchell, and E.L. Chaney. Virtual simulation in the clinical setting: Some practical considerations. *Int. J. Radiation Oncology Biol. Phys.*, 19(4):1059–1065, October 1990.

[Sch88] H. R. Schwarz. *Finite Element Methods.* Academic Press, 1988.

[SGR92] Charles M. Strother, Virgil B. Graves, and Alan Rappe. Aneurysm hemodynamics: An experimental study. *AJNR*, 13:1089–1095, July 1992.

[Shi85] Bhimsen K. Shivamoggi. *Theoretical fluid dynamics.* Martinus Nijhoff Publishers, 1985.

[Shn92] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction.* Addison-Wesley Publishing Company, 1992.

[Ste79] William E. Stehbens. *Hemodynmics and the Blood Vessel Wall.* Charles C. Thomas, Sprinfield Illinois, 1979.

[SVL91] W. J. Schroeder, C. R. Volpe, and W. E. Lorensen. The stream polygon: A technique for 3D vector field visualization. In Gregory M. Nielson and Larry Rosenblum, editors, *IEEE Visualization '91*, pages 126–131, Los Alamitos, California, October 1991. IEEE, IEEE Computer Society Press.

[TGB+90] Russel Turner, Enrico Gobbetti, Francis Balaguer, Angelo Mangili, Daniel Thalmann, and Nadia Magnenat-Thalmann. An object-oriented methodology using dynamic variables for animation and scientific visualization. In T. S. Chua and T. L. Kunii, editors, *CG International '90*, pages 317–328. Computer Graphics Society, Springer-Verlag, June 1990.

[VDLD89] Fernando Viñuela, Jacques Dion, Pedro Lylyk, and Gary Duckwiler. Update on interventional neuroradiology. *AJR*, 153:23–33, July 1989.

[vW92] Jarke J. van Wijk. Rendering surface-particles. In Arie E. Kaufman and Gregory M. Nielson, editors, *IEEE Visualization '92*, pages 54–61, Los Alamitos, California, October 1992. IEEE, IEEE Computer Society Press.

[vWvL93] Jarke J. van Wijk and Robert van Liere. Hyperslice: Visualization of scalar functions of many variables. In Gregory M. Nielson and Dan Bergeron, editors, *IEEE Visualization '93*, pages 119–125, Los Alamitos, California, October 1993. IEEE, IEEE Computer Society Press.

[ZHR+93] Robert C. Zeleznik, Kenneth P. Herndon, Daniel C. Robbins, Nate Huang, Tom Meyer, Noah Parker, and John F. Hughes. An interactive 3d toolkit for constructing 3d widgets. In *Computer Graphics Proceedings*, pages 81–84. ACM SIGGRAPH, ACM Press, August 1993.