Visualizing Glacier Ice Flow in an Immersive Environment

P. Ekman

November 13, 2000

Abstract

This report describes the application of immersive visualization techniques to the problem of visualizing glacier ice flow. An application based on IRIS Performer, pfCAVE and VTK was developed to implement the visualization. A high degree of interactivity is maintained in the application through the use of movable probes controlled by a positional tracker. VTK proves to be a very powerful visualization tool but one that has to be used with care in order to achieve good performance. A major problem with the development of visualization applications for immersive environments is the lack of a general graphical user interface system designed for these environments.

Visualisering av isflöden i glaciärer i en immersiv miljö

Denna rapport beskriver hur isflöden i glaciärer kan visualiseras i en immersiv miljö. En applikation, baserad på IRIS Performer, pfCAVE och VTK, som implementerar visualiseringen har utvecklats. God interaktivitet uppnås genom användandet av positionssensorer kopplade till verktyg i applikationen. VTK visar sig vara ett kraftfullt visualiseringsverktyg som dock måste användas med viss försiktighet om god prestanda skall kunna uppnås. Bristen på generella grafiska användargränssnitt för immersiva miljöer är det problem som utgör det största hindret vid utveckling av immersiva applikationer.

Contents

Chapter 1

Introduction

In the autumn of 1998 I was approached by Peter Jansson, associate professor of glaciology at the Department of Physical Geography, Stockholm University. He was involved in a collaboration with scientists from Eidgenössische Technische Hochschule, ETH, in Zürich where they applied a numerical model of glacier evolution developed in Zürich to Storglaciären, a much studied glacier in the north of Sweden. Jansson was dissatisfied with the quality of visualization in the glaciology area. He wanted to know if I had any ideas on how to visualize the output data from the glacier model. At this time the Center for Parallel Computing, PDC, at the Royal Institute of Technology in Stockholm, was in the process of installing a CUBE, a CAVE-like system for immersive visualization. The intention was to provide the Swedish scientific community with the means to create advanced immersive scientific visualizations. I discussed the problem with Johan Ihrén at PDC and he agreed to get PDC to sponsor the development of a visualization application for the visualization of glacier ice flow in an immersive environment as my masters project. What Peter Jansson was interested in was to see what could be done with the kind of visualization technology available at PDC. If immersive visualization would lead to new insights and whether it would be a useful educational tool. The goal of this work is to provide an interactive visualization application for glacier ice flow utilizing immersive environment technologies to try to answer those questions.

Chapter 2

Glaciology

What is a glacier? A glacier is a mass of ice and snow that is deformed by its own weight and is in motion under the influence of gravity. Glaciers are formed in places where it is cold enough that some or all of the snow that falls there remains frozen throughout the year. Glaciers range in size from small glaciers of a few hundred m^3 of ice to the east and west Antarctic ice sheets that together contain 30 million km^3 of ice — 70% of the planet's fresh water [Näslund, 1998].

Glaciers are studied for a variety of reasons. A glacier has a profound effect on the landscape upon which it rests since it efficiently erodes its bed. Large amounts of debris is picked up by the ice, moved around and finally deposited, creating features such as moraines and eskers. Since large areas of the Earth's surface have been covered by ice during various times in the past understanding how a glacier works is imperative to understand how the landscape of these areas was formed.

Glaciers can also have a more immediate impact on their surroundings. A *jökulhaup* is when a reservoir of liquid water is catastrophically drained through some conduit in a glacier. The reservoir can be formed by normal melt, volcanic activity, or when an advancing glacier dams a stream. In 1996 a large jökulhaup occured in Iceland when a volcano erupted beneath the Vatnajökul icecap. More than 3 km^3 of water was discharged over a period of two days washing out bridges and powerlines, causing damage to the cost of approximately 15 million US dollars [Brandsdòttir, 1996]. Successful prediction of these events would greatly reduce the risk of living or working close to glaciers.

There is much talk about global warming and a fear that the Antarctic ice sheets would melt and cause a significant rise in sea level. In these discussions there are many questions that glaciology can help answer. What will happen to the ice sheets when the temperature rises? When precipitation increases? How fast do the ice sheets react to climatic changes?

The ice that is formed at the surface of a glacier by falling snow can be preserved for hundreds or even thousands of years. This ice contains a record of the chemical composition of the atmosphere at the time when it was first frozen. Old ice recovered from boreholes can thus be used to reveal what the climate was like in the past (paleoclimatology). Glaciers also act as climate indicators. As they react to changes in precipitation and temperature their shape changes. This change can be observed to provide an indication of what is happening to the climate.

2.1 Basic concepts : Storglaciären

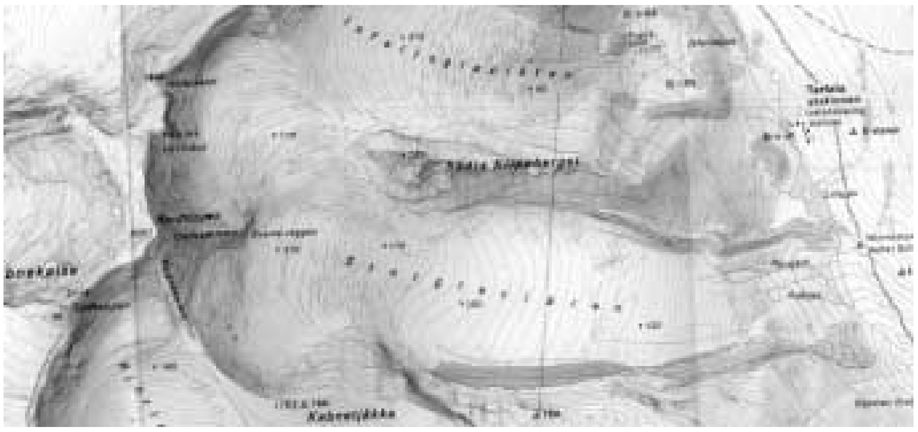

Figure 2.1: Storglaciären.

The glacier chosen to verify the model on is Storglaciären, a small subpolar valley glacier. Storglaciären is located at 67°55'N 18°35'E in the Kebnekaise massif in the north of Sweden. It is 3.2 km long, has a surface area of 3.1 km^2 and an average thickness of 93 m. Storglaciären is one of the most studied glaciers in the world. In 1947 the Tarfala Research Station was founded primarily to support the mass balance measurements started on Storglaciären in 1945 [Jansson and Holmlund, 1998]. This station is located in the Tarfala valley within easy walking distance of several glaciers in the area, including Storglaciären, and it has facilitated much research in the area.

Storglaciären emerges from two basins on the northeast and southeast side of the north peak of Kebnekaise, and an ice tongue extends down to the Tarfala valley. The basins make up the *accumulation* area of the glacier. In the accumulation area the loss of mass, or *ablation*, due to melt is less than the accumulation of mass during the winter. The tongue of the glacier makes up the ablation area. This part of the glacier loses more mass due to melting than is gained by accumulation.

The mechanics of a glacier is described in detail in [LeB Hooke, 1998]. The main source of accumulation is precipitation although drifting, avalanching and condensation also contribute. At the end of the winter season the whole glacier is covered with snow. As the temperatures rises with the onset of summer the snow at the surface starts to melt. The difference in temperature due to altitude causes more ablation in the lower reaches of the glacier. In the ablation area the snow eventually melts altogether, revealing the ice below. As the summer passes the snow line retreats up the glacier. When the melt season ends the snow line defines the boundary between the accumulation and the ablation areas.

The ice is formed in the accumulation area. As the thickness of the snow pack grows the pressure from the overlying snow causes the snow deeper in the pack to start to turn into ice. Snow that is older than a year is called *firn.* As the pressure increases and melt water from the snow surface percolates down and refreezes in the firn layers the firn eventually turns into ice. Gravity causes the ice to flow downwards where the temperature is higher and the ablation is higher than the accumulation. At some point the ice reaches the bare ice surface in the ablation area where it melts.

The evolution of a glacier is governed by its *mass balance*. The mass balance is the amount of mass (expressed as volume of water) lost or gained by the glacier during a year. A positive mass balance means that the total accumulation was greater than the total ablation and the glacier experienced a net growth that year. The mass balance is a function of the climate. A year of low temperatures or high levels of precipitation causes the mass balance to be positive and, conversely, high temperatures or low levels of precipitation cause a negative mass balance. The mass balance averaged over time indicates whether the glacier is advancing or retreating and is also a measure of the climatological trend. The mass balance of Storglaciären has been continously recorded since 1945 and constitutes a unique data resource. The wealth of knowledge about Storglaciären, particularily the mass balance measurements and mappings of the bed and surface topography, makes it a good choice for the verification of the ice flow model.

Ice is, to a good approximation, an incompressible crystalline material. The flow, or strain, of ice is usually taken to be related to the applied stress by Glen's flow law

$$\dot{\varepsilon}_e = (\frac{\sigma_e}{B})^n.$$

Here $\dot{\varepsilon}_e$ is the effective strain rate tensor, σ_e is the effective stress tensor, B is a viscosity parameter and n is a parameter that depends on the dominant deformation mechanism but is commonly considered to be a constant ≈ 3. Glen's flow law can be expanded to take into account effects such as temperature, hydrostatic pressure and crystal orientation.

2.2 The model

The glacier ice flow model was developed by Olaf Albrecht at ETH in Zürich Switzerland [Albrecht, 1999]. The model consists of a glacier mechanics part based on a model developed by Blatter [Blatter, 1995, Blatter and Colinge, 1998], and a glacier surface evolution part developed by Albrecht [Albrecht, 1999]. The model has been used to study the relationship between the geometry of a glacier and its mass balance. The model is controlled by a bed topography, an initial surface topography and a set of mass balances. It calculates the glacier reaction to the given mass balance and generates velocity and stress fields and the calculated ice geometry [Albrecht et al., 1999].

Chapter 3

Visualization

A scientific computational code run on a supercomputer-class machine today commonly produces many gigabytes of data. These data must usually be processed in some way to enable researchers to draw conclusions from it. Scientific visualization is the process of generating images from these kinds of data. Large data sets with three or more dimensions are difficult to visualize effectively using a normal 2D display. Important features and the structural information of the data is frequently lost by the transformation to two dimensions. Even if a 2D picture would be enough to bring out the important parts of the result extracting that picture from gigabytes of data poses a difficult problem.

There are several difficulties with visualizing large datasets. Much extra computation is required to generate graphics from the numerical results. High graphical complexity means that powerful graphics workstations are required to generate the graphics. Efficient algorithms and methods have to be developed to bring out features of interest in the results. One major problem is the difficulty to comprehend and manipulate complex three-dimensional graphics on a two-dimensional display. Effects such as fog can be used to increase the spatial awareness but this may obscure details and hide larger scale patterns. Interaction methods in visualization programs are most commonly based on the keyboard and mouse, but these tools were not designed to work well in three dimensions. Trying to accurately manipulate a 3D-structure with a mouse is often an exercise in frustration.

Using Virtual Reality (VR) technology to do scientific visualization in an immersive environment can alleviate some of these problems. In this text I will take the term *immersive environment* to mean something that involves stereographic real time imaging and an intuitive way of interacting with, and moving around, objects in the virtual space.

The VR equipment that was the target hardware of the developed visualization application consists of three principal parts. A 3D display device, a 3D position tracking system and a host computer. The 3D display systems at PDC utilize so-called *shutter glasses*. Human stereo vision is mainly based on the difference in position between the two eyes, the parallax. The view for the left eye has a different perspective than that for the right eye. The display system mimics this effect. A pair of LCD shutter glasses obscures the vision of one eye at a time. The host computer renders the viewed scene from the perspective of the unobscured eye. Every other frame the glasses switch between obscuring the left and right eye and the host computer is synchronized so that it renders the right frame at the right time. Since two frames have to be drawn for each effective frame in the graphic being displayed the effective frame rate is half that of the actual frame rate. For the systems at PDC the frame rate is 96Hz and the effective frame rate is thus 48Hz.

A 3D position tracking system is employed to be able to intuitively control the applications and to enable the computer to draw the viewed geometry according to the position of the head of the viewer. The tracking systems used at PDC are electromagnetic. A fixed emitter generates a magnetic field that is picked up by small sensors whose position relative to the emitter can then be calculated.

Applications are controlled with an interaction device called a *wand*. The wand is a small stick with three buttons and an analogue joystick with an attached positional tracker. The wand is free-floating, the tracker system feeds the position and orientation of the wand in space to the application and the state of the joystick and the buttons can be tested.

The display devices available at PDC are an ImmersaDesk and a CUBE. They both utilize a combination of back-projection displays and 3D tracking that was pioneered by the Electronic Visualization Laboratory (EVL) at the University of Illinois [Cruz-Neira et al., 1993]. The ImmersaDesk is a back-projected 1.3 m × 1.6 m screen that is tilted at a 45° angle, encased in a large wooden box. The CUBE is based on the same technology as the ImmersaDesk. It consists of six back-projected screens arranged as a 3 m × 3 m × 2.5 m cube with one screen mounted on a door frame. This arrangement gives the user a much greater sense of immersion than with a single screen.

Although there are other types of VR equipment, such as head-mounted displays, the projection based system described above has several advantages. It is easier to get sharp pictures with good resolution from a projection based system. People are often reluctant to put on bulky helmets whereas simple stereo glasses are usually perceived as less restrictive. Several people can share the view without having to duplicate expensive display systems, a crucial property if the system is to be used for education. PDC also has a lot of experience with these types of systems which made development much easier.

The host computer receives tracking data from the position tracking system and sends it to the application program which proceeds to generate the graphics to be displayed. It also makes sure that the shutter glasses are properly synchronized with the graphics refresh. The details of the host computers used with the CAVE and ImmersaDesk are tabulated below.

	CAVE	ImmersaDesk
Machine	SGI Onyx2	SGI Octane
CPU	12 × MIPS R10000@195Mhz	2 × MIPS R10000@250Mhz
Memory	4096MB	1024MB
Graphics	3 × IR	EMXI

Chapter 4

Choosing the tools: COVISE vs VTK

Two visualization systems were considered during the first part of this work: COVISE and VTK. These two systems are architecturally similar in that they both are based on the data flow network paradigm. However they differ in implementation and thus have different advantages and disadvantages.

4.1 Data Flow Networks

In the data flow network paradigm a visualization system is constructed by building a directed graph, the *network*. Conceptually, data flows between the nodes in the network along the edges. The nodes perform operations on the input data and pass it on to their outputs. In both the systems considered, a node in the network is called a module. The network is constructed by connecting the outputs of some modules to the inputs of others. A typical visualization network using this paradigm might consist of a data-read module that read the data to be visualized from file and passes it on to one or several processing modules that might extract the data of interest from the dataset, color it and create the geometry that makes up the visualized result. The last module in the network is commonly a rendering module, a module that opens a window and renders the geometry, but can also be output modules of other kinds, for instance modules that write an image or movie file to disk. The visualization network is also sometimes referred to as the visualization *pipeline* in an analogue to a graphics rendering pipeline.

4.2 COVISE

COVISE stands for Cooperative Visual Simulation Environment. COVISE development was initiated in the Pilot Applications in a Gigabit European Integrated Network (PAGEIN) project, and has continued at the Computer Centre of the University of Stuttgart (RUS) since 1993 [Lang et al., 1997]. Salient features of the COVISE system are support for distributed applications, the Mapeditor for graphical design of the visualization networks and the VR renderer module COVER (COVISE Virtual Reality Environment).

The visualization networks in COVISE are created in the Mapeditor (Figure 4.1), on a graphical “canvas”. Modules are selected from a library and placed on the canvas. Each module has a number of ports that can be connected to other modules. There are three kinds of ports. Data input ports recieve the data for the module to operate on, such as pure application data, geometry or color information. Data output ports output the data generated by the module. Parameter input ports are used to control the operation of the module. A window can be opened with controls for the enabled parameter ports or the ports may recieve their input from other modules. An Application Programming Interface, or API, is provided for writing new COVISE modules.

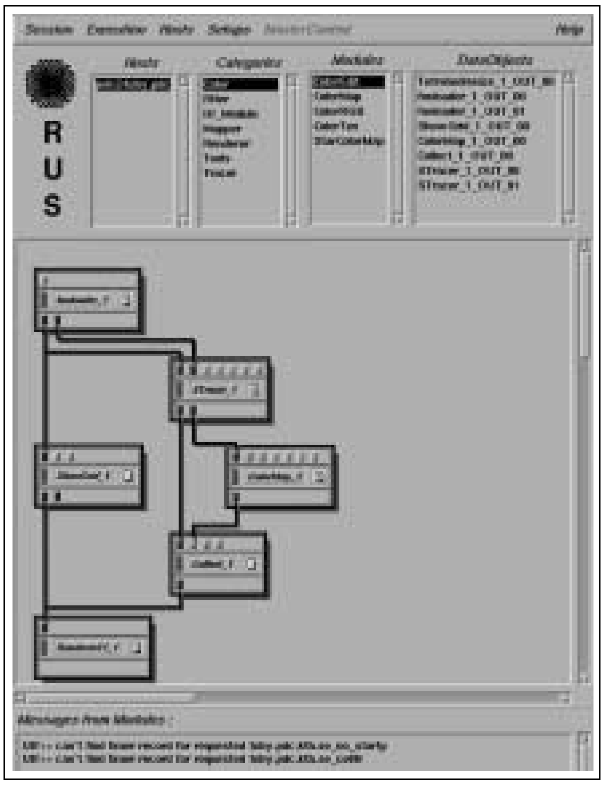

Figure 4.1: The COVISE Mapeditor.

As its name implies COVISE was designed with cooperative visualization in mind. Several instances of COVISE can work on the same data simultaneously. Data are stored in a Shared Data Space (SDS). The SDS will most likely be implemented as shared memory but could concievably be some other data sharing facility. Each SDS is managed by a COVISE Request Broker (CRB). When the application wants to access a data object in the SDS it presents the request broker with a handle, the CRB looks up the object – or creates one if necessary – and returns a pointer to the object in the shared data space. A COVISE application can access data from different shared data spaces transparently. The local request broker takes care of contacting brokers for other data spaces it knows about and transfers data objects between the data spaces as necessary.

COVISE has several standard rendering modules, the three major ones are the Inventor- and Performer-based renderers and the COVER renderer. The application is controlled by enabling parameter inputs in the modules that make up the visualization network. When enabled, inputs get their controls added to the ControlPanel window where they can be manipulated. Most renderers contain controls for manipulating the geometric objects rendered, for moving the viewpoint, and for changing rendering properties such as fog, wireframe rendering, textured rendering etc. The three major renderers all support headtracking and stereo rendering. The COVER renderer can be used with immersive devices such as ImmersaDesks or CAVE's. It supports viewpoint transformations controlled by a 3D-tracker, input devices such as the Polhemus Stylus and the Pyramid Systems wand and multiple display devices such as those employed in a CAVE. An important distinction of the COVER renderer is that it is the only renderer to support integrated control of modules in the visualization network. This is, however, limited to cutting surfaces, isosurfaces and particle traces for unstructured grids.

4.3 VTK

VTK stands for Visualization Toolkit and is a C++ class library for visualization and image processing. VTK was written by Will Shroeder, Ken Martin and Bill Lorensen at GE Corporate Research [Schroeder et al., 1996, Schroeder and Martin, 1999]. The visualization network is constructed by writing code in C++, Java, Python or TCL which means that some programming knowledge is required to write VTK applications. The VTK source code is open and available free of charge although under a license that governs the terms of use. The major features of VTK is the availability of the source code, the fine granularity of the modules, the dynamic development of VTK itself and the textbook The Visualization Toolkit.

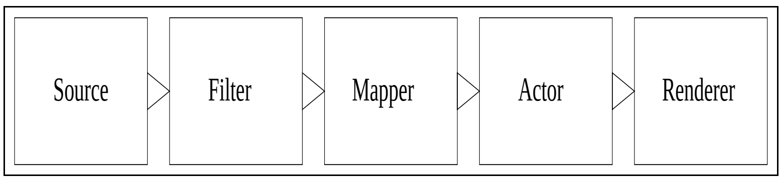

Figure 4.2: The VTK visualization pipeline.

The VTK visualization pipeline (see figure 4.2) starts with a *source* object. A source is either a data reader module or a module that generates data from scratch. The data from a source is passed through *filters* that treat the data in some way. Filters may filter the data or implement some visualization algorithm. The output of the filters – or from the source if the filters are bypassed – is passed to a *mapper* that generates geometry from the input. The geometry generated by a mapper is embedded in an *actor* which contains the geometry and the properties of the geometry such as color, textures, position and so on. Finally the actor objects are added to a *renderer* module that renders the geometry. Changes to parameters in the pipeline are propagated through the modules downstream and cause them to update themselves. Although the data conceptually flows through the network it is usually passed by reference whenever possible.

VTK users are encouraged to write their own VTK modules and make them available to the VTK community. The availability of the source code, the extensive documentation, the large amount of example code and the usefulness of the toolkit has made VTK popular and the number of modules/classes that is part of VTK today is large. Bug fixes, performace improvements and new modules are added to VTK almost daily by both its authors and users. The large number of modules in VTK makes it a flexible tool, but can also be confusing since there are generally many ways to accomplish a certain task and it is not always apparent which one is most efficient.

The Visualization Toolkit [Schroeder et al., 1998] is a general visualization textbook written by the creators of VTK. The book describes the basics of scientific visualization and the algorithms used and uses VTK to illustrate the techniques described. Although the aim of the book is placed firmly on visualization techniques it still constitutes a good introduction to the use of VTK and is a useful index into the multitude of example codes that are supplied with VTK.

4.4 Performer

Performer is a scene graph API developed by SGI (formerly Silicon Graphics). A scene graph is a way of structuring the geometric objects in a scene so that they can be rendered as efficiently as possible. A scene graph is a tree structure with nodes that contain the geometry and some sort of ordering that governs the tree traversal. Any serious graphical application employs some sort of scene graph, be it a highly optimized application specific BSP-tree or a more general structure. The Performer scene graph is designed for visual simulation such as for instance flight simulators and is highly optimized for this task. This makes applications that use Performer to navigate around mainly static geometry efficient at the cost of generality. Performer applications with a high degree of dynamic geometry will perform poorly compared to special purpose scene graphs. The interior nodes in a Performer scene graph contain transformation matrices and various kinds of special purpose functionality such as animation and level-of-detail. The leaf nodes contain the geometry and its state (such as texture and color). The most basic type of geometry node is the *geoset*, it contains a set of homogenous geometric primitives such as points, lines or triangles. Sets of geosets are grouped together in *geode*s which are used to represent a graphical object in the scene.

4.5 CAVElib

The most common API for writing CAVE applications is CAVElib, a C library that wraps around OpenGL. CAVElib was developed at EVL and provides functions that transform the viewpoint so that the viewed geometry is rendered in stereo, functions that transform the viewpoint according to the position of a tracker, and functions that access the buttons and joystick on the Pyramid Systems wand controller. The biggest benefit of using CAVElib is that it is a very thin and low-level layer. It imposes little overhead on the rendering process and thus has limited impact on performance. Dave Pape at EVL has written an API called pfCAVE that embeds the functionality of CAVElib in a form that is easily usable from Performer [Pape, 1997].

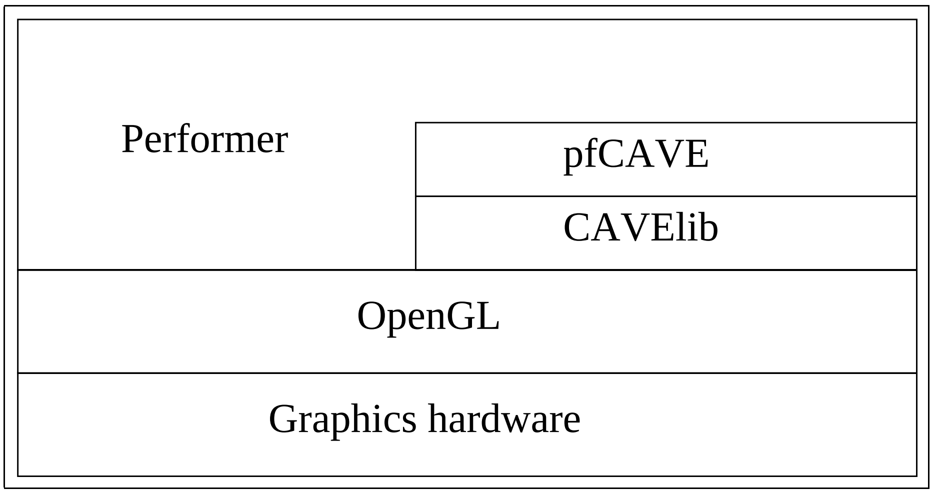

Figure 4.3: Rendering pipeline.

4.6 vtkActorToPF

Paul Rajlich at the National Center for Supercomputing Applications (NCSA), in the United States, has written a VTK module called vtkActorToPF. It converts vtkActors to Performer geodes. Since there was no CAVE renderer in VTK at the time work was begun the VTK-vtkActorToPF-pfCAVE combination was the easiest way to get a visualization up and running in an immersive environment quickly [Rajlich, 1998b, Rajlich et al., 1998].

Chapter 5

Implementation

The problem of visualizing the glacier flow data can be divided into a couple of distinct steps.

- Data format conversion and processing
- Reading the data into the application
- Applying visualization techniques to generate geometry and color from the data
- Rendering the geometry and providing a way of controlling the application

The first point above is independent of the system chosen to implement the visualization application, all others are more or less affected by the APIs and visualization systems used.

5.1 Data format issues

The data from the model is stored in ASCII format. Each row consists of the X (latitude), Y (longitude) and Z (altitude) coordinates and the velocity vector and stress tensor at this point. The dataset is topologically a hexahedron, a structured grid, and regular in the X and Y dimensions but irregular in the Z dimension. The resolution of the grid is 150 m $\times$ 150 m along the X and Y axes and it has 26 layers in the Z direction. All points on the XY-plane have the same number of layers in the Z direction, at points on the grid where there is no ice the Z-coordinates for all layers coincide. The structured format of the data is a good thing, it makes it easy to find neighboring cells by simple index manipulation and the structure information of the data set is implicit; there is no need for special connectivity data structures. For multi-timestep results each timestep is stored in its own file.

I decided to store the data in an intermediate data format to make it easier to adapt to changes in the output data from the model and to make the data less unwieldy. Since the nature of the ice flow data is very similar to that of a laminar fluid flow, it made sense to try to use an established Computational Fluid Dynamics, or CFD, data format as the intermediate format. CFD is a very active research area and there are a number of popular applications and data formats in use. Since there are readers for the most popular data formats already implemented in COVISE, VTK and other visualization systems, using such a format would make it easier to both use the finished application with other data sets and to import the glacier data in other visualization applications. A couple of CFD formats were briefly investigated and one, CGNS, was studied more seriously.

CFD General Notation System (CGNS) is a data format developed by the Boeing Commercial Airplane Group under a NASA contract. CGNS was devised to facilitate the exchange of CFD data between applications and is portable and extensible. An API for storing and accessing CGNS data has been developed but I was unfortunately not able to obtain it at the time. CGNS is very complex because of its generality. It can handle structured, unstructured and mixed topology grids, multi-block data and among other things allow archiving of the equations used to generate the data. [Poirier et al., 1998]

In the end I did not see much point in spending lots of time trying to implement a complex format converter since the data lended itself to a simple structured layout. A straight-forward proprietary data format would be quick to implement and could easily be changed later if necessary. Indeed it later proved to be simple to import the ASCII data to AVS, a commercial visualization application, for verification.

The data format, called Foo data format, or FDF, is described in detail in appendix A. The important features are summarized here. The data is split into three files: a metadata file that describes the contents and dimensions of the data, one or more coordinate files containing the grid for each timestep, and one or more data files containing the data. The metadata file is a human readable ASCII file. The coordinate and data files are binary files. This saves space compared to the original ASCII format and allows much higher I/O rates. The data in the binary files are sequentially ordered to enable the application to read the data using a big stride, thus improving I/O performance when reading large datasets from disk. The binary format is essentially data from memory written as is to disk, this makes reading and writing easy. Unfortunately it also makes the format non-portable because of byte order and word size issues.

A program was written to convert the model output data to FDF. During the development the ordering of the coordinates in the model output data changed. The conversion program thus handles two different orderings: ZYX and ZXY. It cannot figure out the ordering by itself. The conversion program can also change the origo of the dataset. The coordinates in the model output data are given in RR92, "Rikets nät", the national coordinate system for Sweden. In this system the origo is more than 7 million meters south of the grid while the altitude never exceeds 3000 meters. Since the size of the Y (and to a lesser extent the X) coordinate is so large it is prone to round-off errors in the application and a local coordinate system makes it easier to think about where things should be placed.

5.1.1 Data reading

The data read module code is similar between the COVISE and the VTK implementations. The metadata file is read, and a data information structure is set up with the name of the dataset, the dimensions and the number of timesteps. The grid coordinates and data are read into arrays, one for each scalar value (i.e. three arrays for the coordinates, three for the velocity vector and six for the stress tensor). COVISE stores data in this way internally so the COVISE FDF reader module became very simple – it reads the data sequentially from disk, a single read per array, creates a COVISE data object and copies the data into it with a single `memcpy()` call per array. The tensors were not used by the COVISE implementation, nor does it support multiple timesteps.

The VTK reader (vtkFDFReader) is a little more elaborate. This is partly because the data points had to be added one by one and partly because it had to read multiple timesteps as well as the tensor data. Because the grid of one dataset contained dummy altitude values where the input grid of the model was less dense than the computational grid, the VTK reader also does some interpolation. The VTK reader reads the data in the same way as COVISE and then builds up 4D-indices for each array so that the data can be accessed by timestep and XYZ coordinates in the grid. The dummy altitude values were set to 9999, the vtkFDFReader therefore contains a maximum altitude value, anything higher than that is considered a dummy value. When a dummy value is found all surrounding cells in the XY-plane are queried of their altitude, valid altitudes are used to linearily interpolate the altitude of the current cell. If no valid altitudes are found (and this could only happen at the edges with the data set in question) the altitude is set to zero. VTK data objects are then created for the grid coordinates, the velocity vectors, velocity magnitudes and stress tensors, one object for each timestep. The vtkFDFReader module contain methods for stepping backwards and forwards in multiple timestep data. These methods change the output of the module to a new object that contains the data for the timestep requested.

5.2 Creating the visualization

In the VTK implementation each visualization function can be described as the modules between the data reader and the renderer. The set of VTK modules that make up a visualization function, or *method* consists of a probe that determines on what part of the data the function should be applied, one or more modules that manipulate that data, a mapper and an actor that encapsulate the resulting geometry. In most cases the only part of this set of modules one would want to manipulate is the probe module; the probe may be moved or resized. A *visualization entity* is a visualization method instance; the set of VTK modules and the geometry that they generate. A visualization method might be the set of VTK modules that generate a streamline and a visualization entity could then be one actual streamline in the scene together with the modules that generated it. Conceptually it might seem excessive to duplicate modules like this but in practice a module is just the data defining the particular instance. In the code an entity is made up of the probe module and the geode that it generates. The rest of the modules making up the visualization method are kept internal to VTK, they can only be accessed indirectly through the probe module. In this way, given an entity, the program can add or remove the geometry to the scene (through the geode) or manipulate the probe module. All created entities are kept on a linked list until they are deleted.

The visualization methods are implemented as a library of function calls that return entities. The method creation functions are passed parameters defining the properties of the probe (position in the dataset in world coordinates, normal direction and so on), a pointer to the dataset, a color lookup table mapping scalar values to color, the Performer shared memory arena in which the geode is allocated to support multiprocess rendering, and feature parameters for the visualization function.

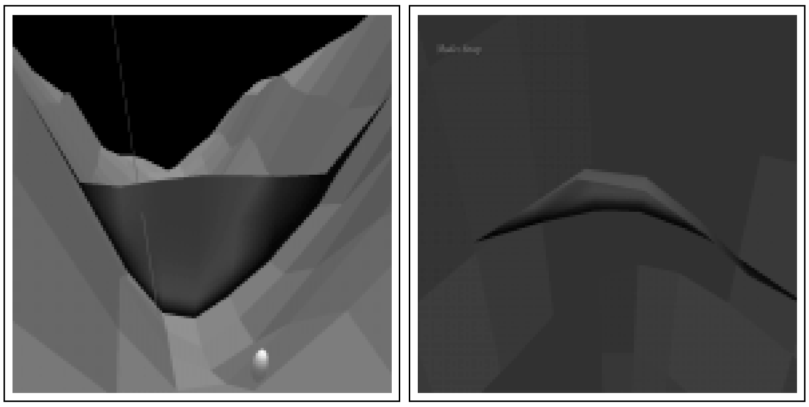

Figure 5.1: A Cutting plane (a) and a warp plane (b).

Scalar data in the datasets is visualized in the same way in the COVISE and VTK implementations, using a plane to cut the data and map the scalar values on the cut to color (figure 5.1a). This technique was chosen because it is easy to implement and the resulting graphic is straighforward to interpret. This was the first try at visualizing data in COVISE and it was immediately apparent that interaction was a problem. There was no way to move the position of the plane and reevaluate the visualization pipeline with the plane at the new position. Geometry could be moved around inside the renderer but there was no mechanism for feeding the positional changes back into the probe module at the start of the network. The way the position of a probe is changed in COVISE is by entering the new coordinates in the parameter window, or the probe can be animated by setting the maximum and minimum coordinates for some axis and let COVISE step through the positions between them sequentially. The COVER renderer does support limited feedback to the network but not for structured grid type datasets. Structured grids can in theory be converted to unstructured grids in COVISE but that did not work well nor did it seem like a good solution to the problem. An even bigger problem was the fact that the feedback mechanism was closed. As I wanted to write my own COVISE modules for the tensor visualization I had to write my own renderer to get interactive control of the new modules and that defeated the biggest benefit of COVISE over VTK — the existing user interface. In VTK, using the entity mechanism detailed above, a change applied to the position of a probe can trigger an update of the part of the visualization network making up the entity.

Vector data can be visualized in a number of ways and five of them were implemented. The simplest vector visualization technique is the *hedgehog*. A hedgehog is when the vector data, either on a probe of some kind or in all of the dataset, is simply plotted as lines or arrows, possibly with their magnitudes scaled by some amount. A hedgehog is easy to generate and has a one-to-one correspondence with the actual data. The disadvantage of a hedgehog is that it may be difficult to draw useful conclusions about the data from it. The sheer mass of vector-lines in a volume clutter the display and obscure each other, although this is somewhat alleviated by using a probe.

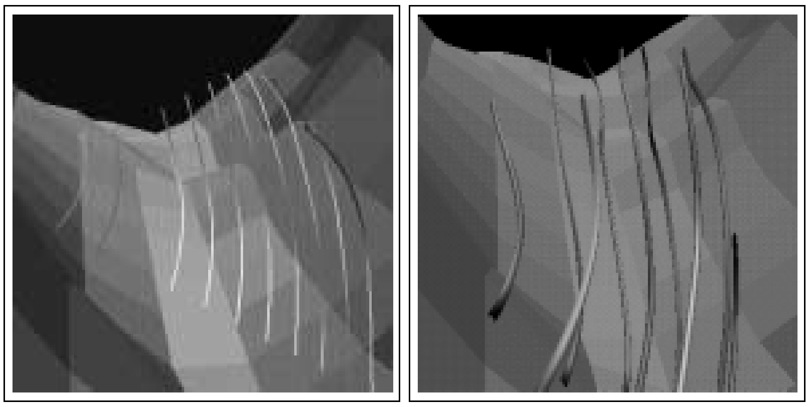
Figure 5.2: Streamlines (a) and streamtubes (b).

Streamlines is another way to capture the properties of vector fields. Conceptually a streamline is the path traced by a massless particle that is released by the probe as it is carried along by the flow defined by the vector field (figure 5.2a). A streamline is generated by moving a predetermined distance, the integration step, from a point in the direction of the vector in that point. The vector at the resulting point is either retrieved directly from the data or interpolated from neighboring points and the process is repeated until either the magnitude of the vectors drop below a threshold or the streamline leaves the dataset.

A form of streamline is the *streamtube*. A streamtube is, as its name implies, a streamline that is represented by a tube (figure 5.2b). The benefit of the streamtube is that the tube can be twisted to represent vorticity. Finally a variant of the cutting plane for scalar data can be used. A plane (or other geometry) is placed in the data and is deformed, or warped, according to the vector field at the intersection (figure 5.1b). This method is called a *warp plane* (no other warping geometry was used here).

Of these methods the only one in addition to the cutting plane tried in COVISE was the streamlines.

Visualizing tensors is much more difficult than visualizing scalar or vector data. The stress tensor is a second order symmetric tensor, the 3 × 3 matrix

$$\begin{bmatrix} \sigma_x & \tau_{xy} & \tau_{xz} \\ \tau_{xy} & \sigma_y & \tau_{yz} \\ \tau_{xz} & \tau_{yz} & \sigma_z \end{bmatrix}.$$

The assumption that ice is incompressible causes the stress tensor to be symmetric. Because of the symmetry the tensor is orthogonally diagonalizable, i.e. it has a set of three linearly independent eigenvectors and can be resolved into three orthogonal vectors. These three vectors are the basis of tensor visualization. The three orthogonal eigenvectors are called the *major*, *medium* and *minor* eigenvectors depending on the size of of the corresponding eigenvalue. The dimension of the tensor makes it hard do visualize effectively. The naive approach, to generate a hedgehog from the eigenvectors, suffers badly from clutter. Color coding the hedgehog with the eigenvalues may help. Another variant of the hedgehog is to use tensor *glyphs*, a geometric object that can adequately represent the information contained in the tensor. A simple tensor glyph is an ellipsoid with the major, medium and minor axes corresponding to the major, medium and minor eigenvectors respectively. The ellipsoid tensor glyphs are used in conjunction with a cutting plane in the implementation (figure 5.3a).

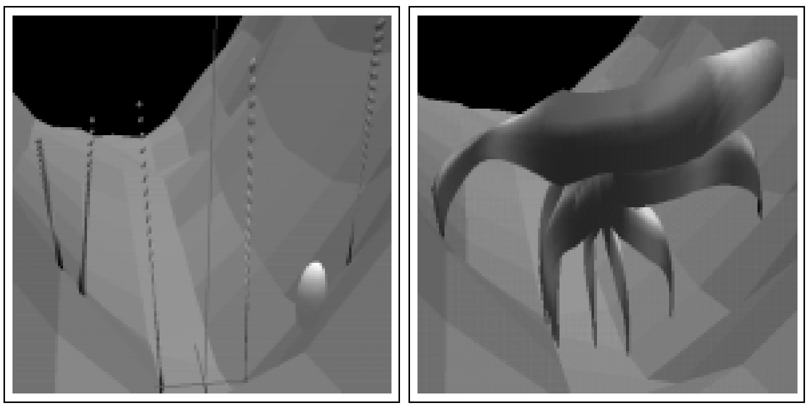

Figure 5.3: Tensor glyphs (a) and hyperstreamlines (b).

A *hyperstreamline* is a generalized streamline for tensor visualization. One of the major, medium or minor eigenvectors of the tensors are treated as a vector field. This field is used to generate a streamline in the same fashion as for a normal vector field. An ellipse perpendicular to the streamline and with its major and minor axes corresponding to the remaining two eigenvectors of the tensor is then traced along the streamline forming a deformed stream tube, a hyperstreamline (figure 5.3b). If for instance the major eigenvector is used for the streamline the trajectory of the streamline indicates the direction of the largest stress component.

5.3 Application framework

The inital experiments with VTK were performed using the standard VTK renderer which is based on Inventor, a SGI scene graph API preceeding Performer. It is very easy to put together visualizations using VTK and the standard renderer, but as in the COVISE case interaction was a problem. Geometric objects (actors) rendered by the VTK renderer can be moved around and the renderer can register a user-defined function to be called on some event. What I tried to do was to move for instance a cutting plane in the renderer and then call the user-defined function to retrieve the new position of the plane actor in order to reevaluate the cutting plane there. This manifested a bug somewhere in VTK that caused the plane actor to move twice, once by the user and again when the plane was reevaluated. The result was that the cutting plane ended up in a place where the data that it visualized was not. At this point I decided to move to Performer and vtkActorToPF.

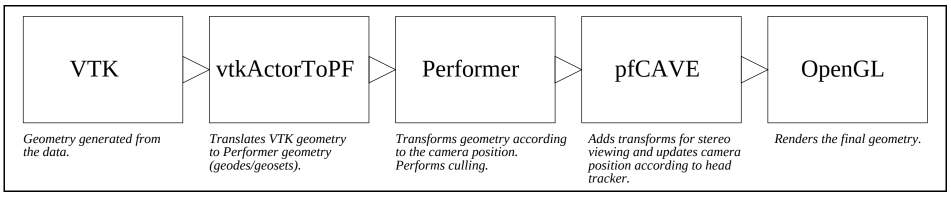
VTK
vtkActorToPF
Performer
pfCAVE
OpenGL
Geometry generated from the data.
Translates VTK geometry to Performer geometry (geodes/geosets).
Transforms geometry according to the camera position. Performs culling.
Adds transforms for stereo viewing and updates camera position according to head tracker.
Renders the final geometry.

Figure 5.4: Application dataflow.

The structure of the application is shown in figure 5.4. The application framework grew out of the testing code and was restructured several times in order to make it as simple and understandable as possible. The application framework needs to provide a way to navigate through the dataset, to move probes and to access functionality in the application. The structure of the application scene graph is showed in figure 5.5. A DCS is a Performer node that encapsulates a transformation matrix. A SCS is essentially the same thing as a DCS with the difference that the matrix of a SCS cannot be changed. This allows the Performer implementation to optimize the scene graph by applying the SCS transformation to the underlying nodes only once. The scaleSCS applies a constant scaling transformation to all nodes in the scene. The navDCS transforms the scene as the user moves around in it. The cameraDCS is used to scale the transformations that move the world and the cursor. This may be necessary since the user may want to move for instance the cursor very fast while it's very confusing if the viewpoint is radically changed when the users head is moved slightly. The cursorDCS controls the position of the cursor and the worldDCS is used to move through the scene.

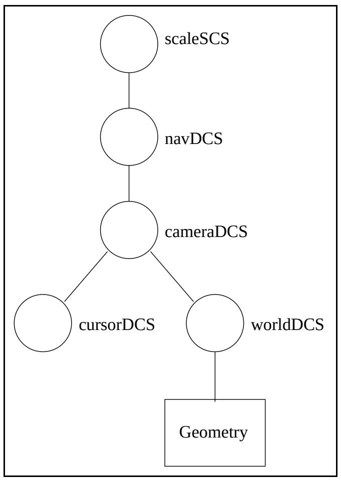

Figure 5.5: Scene graph structure.

The navigation model employed is one where the "world", in this case the glacier, is attached to the wand. When in world-move mode the world geometry is moved in the same direction as the wand, possibly with some scaling applied. Other navigation models, such as flying or walking through the world, are possible and would be easy to add to the application.

Probes are manipulated by means of a cursor. The cursor consists of six lines making up three axes. One of the lines is colored red to signify direction, a plane generated at the cursor position has its normal in the direction of the cursor. The movement model for the cursor is the same as for the world. The cursor is independent of the world, i.e. if the world is moved the cursor remains in the same position relative to the user. Another cursor type that could be employed is that of a "light sabre" where a line or tube is drawn from the wand to the cursor focal point. This kind of cursor works well in an immersive environment but can be ambigous when used with a 2D display. While the ambiguity may still be present with the axis type of cursor described above it can be alleviated by extending the axis lines beyond the bounds of the data. The points where the lines intersect geometry then give a useful depth cue. A method that could conceivably be used to further help resolve ambiguity in the cursor orientation would be to place cones on the axes, oriented so that they all point to the cursor focal point. The occlusion of the cones would give an indication of the cursor orientation, although this type of cursor might clutter the display.

The user interface has caused the most difficulties in this work. The Pyramid wand has three buttons and a joystick. The joystick is hard to use since it is very sensitive and located where it is difficult not to accidentally move it when pressing buttons. The user interface implemented is admittedly not a very good one, it was put together quickly since developing a graphical user interface (a GUI) for immersive environments was not a part of the specification. The left button, when depressed, enables "world move" mode, the middle button enables "cursor move" mode and the right button is the "action" button. Visualization functions are selected by pressing the middle and right buttons simultaneously. As more functions were implemented it became necessary to give a cue to the user as to what function was selected. Finding a workable way to render text took some experimenting, particularly since moving to a multiprocessor machine meant that Performer forked the rendering process from the main application causing some shared-memory headaches.

It turns out that the poor user interface significantly limits the potential of the application. Adding functionality such as for instance isosurface generation to the application is architecturally very simple but the lack of a GUI makes it difficult to control effectively. It would be awkward to change the isovalue interactively without some sort of slider or equivalent GUI component. Selecting visualization methods is also a bit awkward. The user potentially has to cycle through all but one of the implemented functions to get to the desired one and it is all to easy to accidentally execute the current function when trying to change modes.

There is an obvious need for an immersive environment GUI API. The *pfuGUI* API developed for the `perfly` Performer showcase application shows some promise in this respect. Although it is a pure 2D GUI it sits well in a Performer application and is flexible enough that it could be controlled by something other than a mouse, for instance a wand.

5.4 Performance issues

The performance, speed in this case, of the code is an important issue. In order for the application to be really useful, interactivity must be preserved even on large datasets. Implementing a particular visualization function in VTK has to be done with care. The large number of available classes often means that there are many ways to achieve the desired functionality but the performance of the result may vary greatly depending on the classes used.

Slicing the data with a plane that is non-orthogonal to the grid is an expensive operation since the data at most or all points on the plane must be interpolated. Doing orthogonal slicing is much faster but turns out to be problematic with the datasets used here. Since the dimensions of the cells in the dataset are at best 150 m$\times$150 m$\times$10 m the cells are very thin. The limited resolution in the XY-plane makes pure orthogonal slicing less useful. This problem can be alleviated somewhat by precomputing an increased-resolution dataset by interpolation, or if a more regular grid is used.

The flatness of the cells also causes difficulties for streamlines. The integration step has to be chosen carefully. The step is given as a fraction of the cell size measured as the cell diagonal. Unless the step is very small, or if the velocity vectors are very nearly parallell to the XY-plane, it will cause the VTK integrator to step through several cells in the Z-direction in a single step causing instability in the integration. The small integration step necessary is unfortunate because it increases the time spent calculating a streamline.

An area where much performance could be gained is in multiprocessing. Performer will fork off separate processes to do culling, intersection testing and rendering if it will benefit performance (i.e. if the machine has more than one available processor). CAVElib will also fork off processes when doing multi-wall rendering such as in a CAVE. However, the visualizations presented here are in most cases CPU-bound and will not in general benefit significantly from these parallelizations (since they only speed up the rendering process). The visualization network would have to be explicity parallelized in order to improve performance by multiprocessing. The VTK visualization pipeline lends itself well to shared-memory multiprocessing and with some care the visualization pipeline should parallelize quite nicely (see [Rajlich, 1998a] for an example).

5.4.1 Performance experiments

Three special test versions of the application were written in order to study the performance of the code. The three test programs are called streamTest, cutTest and timestepTest. StreamTest creates 20 velocity streamlines. CutTest creates 10 cutting planes transversal to the main flow direction mapping speed to color. TimestepTest creates 10 streamlines and 10 cutting planes and then steps through 5 timesteps. An empty test was also run where the application loads a dataset, opens a window, renders the grid and then terminates. This was done to get a feeling for the magnitude of the startup time. The 1961 data set was used for streamTest and cutTest and the 1961-1965 datasets were used for timestepTest. The performance experiments were conducted with SpeedShop, a collection of tools that can be used to analyse the behavior and performance of programs on SGI systems. The SpeedShop components used were `ssusage(1)`, `pixie(1)`, `ssrun(1)` and `prof(1)`. The `ssusage(1)` program collects execution time and resource usage statistics for an application. The `pixie(1)` program measures code execution frequency on the instruction level by code instrumentation. The `ssrun(1)` program runs performance experiments on an application and collects the resulting data. The raw data collected by `ssrun(1)` is condensed into a readable report by `prof(1)`. All experiments were run in the CAVE simulator on an SGI Octane with 2 250Mhz R10000 CPU's and 1GB of memory.

Execution times and resource usage were measured with the `ssusage(1)` tool. All `ssusage` tests were run five times and the results were averaged. The test programs were compiled with VTK 2.2, VTK 2.4, VTK 2.4 with IPA optimizations, and VTK 2.4 with profiling feedback. The VTK 2.2 and VTK 2.4 experiments were compiled without debug information and with level 3 optimizations. Inter-Procedural Analysis, or IPA, is a collection of algorithms that are used for global code optimizations. The program code is analyzed across procedure calls, something that is not done by normal optimization techniques. The IPA implemented by the SGI compilers do procedure inlining, constant propagation, dead function- and variable elimination and global name optimizations. In the profiling feedback experiment execution data was collected by instrumenting the code with `pixie(1)` and tracing (or profiling) an ideal run of the program with `ssrun(1)`. The compiler can then use the data collected to generate a version of the program that is optimized specifically for the traced run. Profiling feedback data mostly help the compiler to rearrange code to improve the hit ratio of the processor branch prediction logic.

	Elapsed time (s)			
Experiment	VTK 2.2	VTK 2.4	VTK 2.4 + IPA	VTK 2.4 + feedback
cutTest	4.35	1.45	1.43	-
streamTest	2.72	1.10	-	-
emptyTest	0.45	0.28	-	-
timestepTest	26.68	8.91	8.93	8.79

Switching from VTK 2.2 to 2.4 caused a speedup of 3.0 for the cutTest. StreamTest got a speedup of approximately 2.5 and the timestepTest got a speedup of approximately 3. The IPA optimizations slightly improved performance on cutTest but actually decreased it on timestepTest. Profiling feedback only improved performance marginally which indicates that the code either has a limited number of branches (which is unlikely), spend many iterations in most loops (thus amortizing the cost of a mispredicted branch) or that the CPU is mostly successful at predicting branches.

Procedure call statistics were collected by `ssrun` and `prof` for the VTK 2.2 and VTK 2.4 timestepTests. These statistics were collected for an ideal run of timestepTest. The caliper point feature of the SpeedShop tools were used to exclude the startup time of the application from these tests. Caliper points are markers that can be placed in the code. Caliper points allow the profiling tools to gather data only for the parts of the code between specified points. The VTK 2.2 test spent 75% of the time cutting the data while only 6% was spent creating the streamlines. The rest of the time (12%) was mostly spent generating surface normals. The VTK 2.4 test spent 67% of the execution time cutting the data, 6% creating streamlines and 12% generating surface normals.

Major improvements have been made to vtkCutter in VTK 2.4. In the VTK 2.2 test 23% of the execution time was spent in the vtkScalars class, mainly called from vtkCutter. These calls have been almost completely eliminated and account for only 2% of the execution time in the VTK 2.4 test. The vtkFloatArray class have also been improved. The number of calls to methods in vtkFloatArray have been almost halved and the time spent there have been quartered.

It seems probable that the loop length is fairly short but easily predictable by heuristic methods for this type of code. This would explain why the profiling feedback optimizations did little to improve performance. Data at the low level of the application is based on the vtkFloatArray class. All filters and mappers access the data through this class. This means that the performance of vtkFloatArray affects the performance of every part of the application (except for the pure geometry transformation and rendering operations). Most of the speedup gained by the switch from VTK 2.2 to VTK 2.4 is probably due to the improvements in vtkFloatArray. This theory is born out by the fact that the relative execution times of the major parts of the test applications (cutting, streamline integration and normal calculations) remain more or less constant between the different VTK versions.

5.5 Qualitative analysis

The model does not take basal sliding into account, this means that the velocity at the bed is zero. When a streamline is traced backwards in time and passes very close to the bed the zero velocity cells will cause it to stop. This may give the false impression that the traced particle originated at the bed. Since all ice is generated at the surface of the accumulation area all streamlines that are integrated backwards in time should originate there.

The tracker system is not very stable, the tracker position changes noticeably which in some situations can make it difficult to position a probe with precision. This is probably due to a bug in the probe positioning code, or (less likely) a problem with the hardware.

A strange clipping bug has also manifested itself in the application; a cutting plane will sometimes be partly occluded at the bed by parts of the polygons that make up the bed geometry. This effect disappears when the viewpoint is moved close to the anomaly which makes it likely that the problem is due to the limited resolution of the Z-buffer (so called *Z-fighting*).

Chapter 6

Future Work

The most useful addition to the application would be a graphical user interface as this would enable a host of new functionality to be implemented. Being able to move for instance a cutting plane through the data, having the cutting plane automatically reevaluated at each point, would be a good way to increase the interactivity of the application. This would have to be coupled with performance enhancements since the current implementation of cutting planes is rather slow. Among the performance increasing measures that could be implemented are: to modify any abstract data classes used to concrete classes to avoid function calls through virtual function pointers, to inline frequently used methods in those classes, and to parallelize the VTK visualization pipeline. Other useful features would be the ability to "play" through a sequence of timesteps and to be able to create a movie of the visualization. The ability to save the state of the application in order to be able to restart it with relevant visualizations already in place would also be useful.

Chapter 7

Conclusions

The three-dimensional nature and spatial distribution of the glacier ice flow data makes it suitable for immersive visualization even though the datasets are small. VTK has proved to be a flexible and powerful tool although it has to be used with some care to achieve good performance. As the performance of VTK improves rapidly it is becoming viable as a tool for interactively visualizing medium sized datasets (tens to perhaps a few hundreds of MB). The application framework developed could be used to implement a variety of immersive visualizations based on VTK, given some improvements, the most obvious being that of a better user interface. The ice flow visualization is effective and could well be used to study features of the flow, particularly for educational purposes. It is, however, of limited availability since the equipment required for the immersive part is expensive and relatively immature. Using the immersive technologies to bring out the interesting features of the flow data and then create pictures or movies of the visualization that could be shown in for instance a classroom is perhaps a workable compromise.

Chapter 8

Acknowledgements

I would first and foremost like to thank Peter Jansson for his help and for initiating this work; it has been a most interesting project. Thanks to Johan Ihrén for getting PDC to sponsor this work and for agreeing to be my supervisor and to Olaf Albrecht for answering my many questions concerning the numerical model. I also extend my gratitude to Johan Danielsson, Mattias Claesson, Gunnar Ledfelt and Erik Engquist for helping me getting my frequently derailed train of thoughts back on track. Finally, thanks to Johan Norin, with whom I had an interesting discussion during a day of ablation stake measurements on Storglaciären in the summer of 1997 that got me thinking I wanted my master's project to be related to the Tarfala Research Station.

Bibliography

[Albrecht, 1999] Albrecht, O. (1999). *Dynamics of Glaciers and Ice Sheets: A Numerical Model Study.* PhD thesis, Eidgenössische Technische Hochschule, Zürich.

[Albrecht et al., 1999] Albrecht, O., Jansson, P., and Blatter, H. (1999). Modelling glacier response to measured mass balance forcing. Paper to be published in Annals of Glaciology.

[Blatter, 1995] Blatter, H. (1995). Velocity and stress fields in grounded glaciers: A simple algorithm for including deviatoric stress gradients. *Journal of Glaciology*, 41(138).

[Blatter and Colinge, 1998] Blatter, H. and Colinge, J. (1998). Stress and velocity fields in glaciers. part I. Finite difference schemes for higher order glacier models. *Journal of Glaciology*, 44(148).

[Brandsdòttir, 1996] Brandsdòttir, B. (1996). Subglacial volcanic eruption in Gjàlp, Vatnajökull, 1996. `http://www.hi.is/~mmh/gos/`. Valid October 4, 2000.

[Cruz-Neira et al., 1993] Cruz-Neira, C., Sandin, D. L., and DeFanti, T. A. (1993). Surround-screen projection-based virtual Reality: The design and implementation of the CAVE. *Proceedings of SIGGRAPH '93 Computer Graphics Conference*, pages 135–142. `http://www.evl.uic.edu/EVL/RESEARCH/PAPERS/CRUZ/sig93.paper.html`. Valid October 4, 2000.

[Jansson and Holmlund, 1998] Jansson, P. and Holmlund, P. (1998). Tarfala research station. `http://www.geo.su.se/naturgeo/glaciologi/Tarfala/Tarfala.htm`. Valid October 4, 2000.

[Lang et al., 1997] Lang, R., Lang, U., Nebel, H., Rainer, D., Rantzau, D., Wierse, A., and Wössner, U. (1997). *COVISE User's Manual.* University of Stuttgart Computer Centre.

[LeB Hooke, 1998] LeB Hooke, R. (1998). *Principles of Glacier Mechanics.* Prentice Hall, New Jersey.

[Näslund, 1998] Näslund, J.-O. (1998). *Ice Sheet, Climate, and Landscape Interactions in Dronning Maud Land, Antarctica.* PhD thesis, Stockholm University, Stockholm.

[Pape, 1997] Pape, D. (1997). pfCAVE CAVE/Performer library. `http://evlweb.eecs.uic.edu/pape/CAVE/prog/pfCAVE.manual.html`. Valid October 4, 2000.

[Poirier et al., 1998] Poirier, D., Allmaras, S. R., McCarthy, D. R., Smith, M. F., and Enomoto, F. Y. (1998). *The CGNS System.* The American Institute of Aeronautics and Astronautics.

[Rajlich, 1998a] Rajlich, P. (1998a). `http://hoback.ncsa.uiuc.edu/~prajlich/vtkActorToPF/future.html`. Valid October 4, 2000.

[Rajlich, 1998b] Rajlich, P. (1998b). An object oriented approach to developing visualization tools portable across desktop and virtual environments. Master's thesis, University of Illinois. `http://monet.astro.uiuc.edu/~prajlich/T/bigT.html`. Valid October 4, 2000.

[Rajlich et al., 1998] Rajlich, P., Stein, R., and Heiland, R. (1998). `http://hoback.ncsa.uiuc.edu/~prajlich/vtkActorToPF`. Valid October 4, 2000.

[Schroeder and Martin, 1999] Schroeder, W. and Martin, K. (1999). *The vtk User's Guide.* Kitware.

[Schroeder et al., 1998] Schroeder, W., Martin, K., and Lorensen, B. (1998). *The Visualization Toolkit : An Object-Oriented Approach to 3D Graphics.* Prentice Hall, New Jersey, second edition.

[Schroeder et al., 1996] Schroeder, W. J., Martin, K. M., and Lorensen, W. E. (1996). The design and implementation of an object-oriented toolkit for 3D graphics and visualization. *Proceedings of Visualization '96.*

Appendix A

Foo Data Format specification

A dataset has a name, all files belonging to the dataset share this name but with different extensions. The files that make up a dataset are the metadata file, one or more coordinate files and one or more data files. All binary data are stored in big-endian byte order.

The organization of the data files is specified in the metadata file. The metadata file is an ASCII file with the .meta extension. Each line in the file is terminated by a newline. The items in the metadata file are:

```
Name: <255 character string>
X_count: <32 bit integer> # This is the number of nodes in the X
  # direction
Y_count: <32 bit integer>
Z_count: <32 bit integer>
Timesteps: <32 bit integer> # Number of timesteps in the data.
    # This number must match the number of data and
    # coordinate files.
Scalar: <name (64 characters)> <type: int | long | float | double>
Scalar: ...
...
Vector: <name (64 characters)> <size in elements> <type: int | long |
float | double>
Vector: ...
...
Tensor: <name (64 characters)> <rank> <size of vectors in elements>
<type: int | long | float | double>
Tensor: ...
...
```

These items must appear in this order and the first five are mandatory.

The coordinates are stored in files with the extension .crd.*step* where *step* is the timestep number of the coordinate set. There is one coordinate file for each timestep. Each coordinate is stored in its binary representation. The coordinates are organized in three blocks, one for each dimension. The x-coordinates come first, followed the y and z coordinates.

The data are stored in files with the extension .dat.*step*. All data values are stored in the binary representation of the data type specified in the metadata file. Scalar data are stored first in the file in the order specified in the meta data file. Vector data follows the scalar data. The components of a vector are stored together in sequence. Tensor data follows the vector data. The tensors are stored similarily to the vectors.