VizFlow: A Visualization Package for Multi-Block Multi-Processor CFD and MHD Data

George Stanchev
Steven Zalesak
Anil Deane
Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA’s scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA’s institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- **TECHNICAL PUBLICATION.** Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA’s counterpart of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.

- **TECHNICAL MEMORANDUM.** Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- **CONTRACTOR REPORT.** Scientific and technical findings by NASA-sponsored contractors and grantees.

- **CONFERENCE PUBLICATION.** Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or cosponsored by NASA.

- **SPECIAL PUBLICATION.** Scientific, technical, or historical information from NASA programs, projects, and mission, often concerned with subjects having substantial public interest.

- **TECHNICAL TRANSLATION.**

  English-language translations of foreign scientific and technical material pertinent to NASA’s mission.

Specialized services that complement the STI Program Office’s diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results . . . even providing videos.

For more information about the NASA STI Program Office, see the following:

- E-mail your question via the Internet to help@sti.nasa.gov
- Fax your question to the NASA Access Help Desk at (301) 621-0134
- Telephone the NASA Access Help Desk at (301) 621-0390
- Write to:
  NASA Access Help Desk
  NASA Center for AeroSpace Information
  7121 Standard Drive
  Hanover, MD 21076–1320
VizFlow: A Visualization Package for Multi-Block Multi-Processor CFD and MHD Data

George Stanchev
University of Maryland, College Park, MD

Steven Zalesak
Naval Research Laboratory, Washington, DC

Anil Deane
University of Maryland, College Park, MD
NASA Goddard Space Flight Center, Greenbelt, MD
Contents

1 Introduction
   1.1 VizFlow Capabilities ............................................. 3

2 Development Summary
   2.1 Data Conversion Filter ........................................... 8
      2.1.1 The BATSRUS Data Conversion Filter ....................... 8
      2.1.2 The VizFlow Filter ........................................ 8
   2.2 The IRIS GL to OpenGL/GLUT transition ......................... 10
      2.2.1 Porting Commands with Exact Equivalents .................. 10
      2.2.2 Porting Commands Whose Equivalents Are Not Exact ....... 11
      2.2.3 Redundant IrisGL Commands ................................ 14
      2.2.4 Porting the Windowing Interface .......................... 14
   2.3 Fortran 77 to Fortran 90 transition ............................ 19
      2.3.1 Fortran 90 implementations of OpenGL/GLUT ............... 19
      2.3.2 Caveats .................................................. 20

3 Functional Development .............................................. 23
   3.1 Spherical Data .................................................... 23
      3.1.1 NURBS Basics .............................................. 23
      3.1.2 Modeling Curvilinear Polygons with NURBS ............... 27
      3.1.3 OpenGL’s NURBS Interface ................................ 30
   3.2 Rotations .......................................................... 32
      3.2.1 Quaternion vs. Axis-defined Rotations .................... 32
      3.2.2 Trackball Simulation ..................................... 33
      3.2.3 Trackball Modification .................................... 35
      3.2.4 Rotation Around the Domain’s Center Point ............... 37
   3.3 Viewing Transformations ......................................... 38
      3.3.1 The Local View Frame .................................... 39
      3.3.2 Domain Fits the Viewport ................................ 39
   3.4 Polygon Offset .................................................... 39
   3.5 Selection Mode ..................................................... 41
   3.6 Menus ............................................................. 43
   3.7 Performance Issues ............................................... 44
      3.7.1 Display Lists .............................................. 44
      3.7.2 Efficient Update Mechanism ............................... 45
4 User’s Guide

4.1 Getting Started ......................................................... 47
4.2 Modes of Operation .................................................... 47
4.3 Navigation and Object Manipulation ................................. 48
4.4 Displaying Grids ......................................................... 48
4.5 Slices ................................................................. 49
4.6 Status Bar ............................................................... 49
4.7 Saving Data ............................................................. 49
4.8 VizFlow Data ............................................................. 50
  4.8.1 Internal Data Format .............................................. 51
  4.8.2 AMRFCT Format ................................................ 52
4.9 Movie Capabilities ..................................................... 52
4.10 Tracing Field Lines .................................................... 53
4.11 Summary of Command Key Shortcuts .............................. 54
Chapter 1

Introduction

1.1 VizFlow Capabilities

Summary: VizFlow is a flexible visualization package that allows for visualization of two- and three-dimensional fluid dynamics and magneto-hydrodynamics (MHD) data that is adaptively-refined (AMR), multi-blocked, and multi-processor. It supports data in BATSRUS (U/Mich), AMRFCT (essentially PARAMESH (NASA/GSFC) format) and NEKTON (Argonne National Labs.) formats. The data can be in Cartesian or Spherical coordinates. The package is downloadable from the Laboratory for Computation and Visualization, Institute for Physical Science and Technology (www.lcv.umd.edu).

Example 1: BATSRUS is a University of Michigan code that is particularly well suited to modeling the interaction of the solar wind with planetary magnetospheres. It is a state-of-the art code for this purpose and has been a Grand Challenge code for the Computational Technologies Project of NASA/GSFC. The code is a 3D cartesian code that solves the MHD equations in conservation form. It considers block refined data structures with adaptive refinement and can run in a parallel environment. Its output consists of three types of data. Full-field data, consisting of density, momenta, pressure and magnetic fields at various times; Restart data which consists of such data for the final time step as well as block interconnectivity information for restart purposes; and ionospheric interpolated data with ionospheric potential and other data relevant to the ionosphere. For our purposes of visualization with VizFlow we consider only the full-field data, which is the most relevant.

Figure 1.1 shows a typical simulation performed with BATSRUS. Note that the Earth is not shown (i.e. there is an inner boundary where boundary conditions are applied but depicted in VizFlow), but the entire computation is shown. The lines depict block boundaries each consisting of $(4 \times 4 \times 4)$ cells. Three cut planes in each of the directions are shown.

VizFlow allows the rotation, zoom and pan of the view, as well as the choice of interpolation, cell or block boundaries and the various data fields.

Example 2: Spherical Coordinates. The solar wind modeling codes SOLWND and HELIOS can use spherical coordinates. Thus data is generated that while canonically rectangular (hexahedral), needs to be suitably interpreted as lying in spherical coordinates. Figure 1.2 shows an un-interpolated (cell-averaged) values of the density field for the HELIOS code. This is actually
Example 3: AMRFCT is a 3D MD code using PARAMESH/NASA/GSFC block-refined AMR library. The code considers cartesian or spherical coordinates. The code primarily is used for modeling the solar wind turbulence and heliospheric magnetic field structure problems. Figure 1.4 shows an interpolated (cell-averaged) values of the density field for the AMRFCT code for a jet under an externally applied magnetic field. This is a multiprocessor run (i.e. the calculation was performed on several processors).

Example 4: Magnetic Field Lines Topology is often crucial to understanding the dynamics. Small inaccuracies in computation which are otherwise of seemingly of little consequence can have profound influence on the topology of the magnetic field lines and subsequent dynamics. VizFlow allows the computation of magnetic field lines superposed on the other views. Figure ?? shows magnetic field lines computed and displayed by VizFlow. No connectivity information of the blocks is used from the simulation. Instead VizFlow generates such information based on block coordinates.

Example 5: Most features of VizFlow are accessible via hierarchical menus, although keyboard shortcuts are available. Figure 1.5 shows a typical screenshot of the menus.

obtains fast computations in a parallel domain. We use the code of P. Fischer/Argonne National Laboratory to perform simulations of convection, one example case is shown in Figure 1.6. The high order interpolation intrinsic to the code (Gauss-Legendre-Lobatto) is used within VizFlow.

**Other features and capabilities include:**

- View selection
- Cut plane selection
- Color Bar on/off
- Coordinate axes on/off
- Save/restart state
- Movie
- Grid lines (cell, block) on/off
- Screendump
Figure 1.3: AMRFCT data showing an MHD jet. There is a vertically imposed external magnetic field. The lines indicate block boundaries of 4x4x4 cells each.

Figure 1.4: Example showing the magnetic field line tracing capabilities of VizFlow. Shown is the MHD jet flow of the Figure 1.4 with the vertically imposed external magnetic field.
Figure 1.5: VizFlow allows menu driven features.

Figure 1.6: VizFlow supports spectral-element (high-order polynomial interpolant finite-element) data from code *nekton*. Shown is the temperature field of Rayleigh-Bénard convection in a rectangular container, with insulating sides and hot bottom and cold top. 256 3D elements of order 8 were used in the calculation.
Chapter 2

Development Summary

2.1 Data Conversion Filter

Summary: A filter, written in a combination of Fortran 90 and Perl, is used for converting data from BATSRUS format into VizFlow’s native format. Other formats such as Nekton and AMRFCT (which uses the block description of PARAMESH) are automatic within VizFlow. A great deal of flexibility is available in order to accommodate a variety of data sets; for example, parameters like the number of data files, header length of each data file, total number of blocks in a data set, and so on, are calculated automatically by the filtering module. The only input needed from the user is the location of the data set.

2.1.1 The BATSRUS Data Conversion Filter

The data conversion filter is set up to do the following:

- Read a collection of files, each containing BATSRUS data output by a separate processor.
- Within each file, read the data corresponding to the individual blocks comprising the solution domain.
- From the boundaries of the individual cells within each block, compute the block’s center point and size. Along with the variables associated with each cell within a block, write these into a global data structure.
- After all input files have been read, output the global data structure to a single file which can then be imported in VizFlow.

An earlier implementation assumed that the entire file set resided in the same directory, and that parameters such as the number of cells per block, the number of variables associated to a cell, etc. were fixed. Also, all file names were “hardwired”, so in order to read a different file set, one had to modify and recompile the program.

2.1.2 The VizFlow Filter

Since Fortran’s operating system interface is rather clumsy, we have added a preprocessing module, which browses through a data set and computes a number of parameters that are then
passed on to the actual filtering program. The preprocessing module is written in Perl - a scripting language that not only allows for much easier directory and file access but also has superior string manipulation capabilities. At the same time it should be noted that Perl lacks the numerical precision needed for handling CFD data, so writing the entire conversion filter in Perl is impractical. The preprocessing module does the following:

- Parses command line arguments such as the location and the identification string of a data set. For example:
  ```
  convert 710 /some/where/BATSRUS_DATA/
  ```
  will look into the directory `/some/where/BATSRUS_DATA/` and scan all files whose names have the following format:
  ```
  aIO_n00710_pe0000.dat, aIO_n00710_pe0001.dat, etc.
  ```
  The “pe”-string denotes the number of the processor from which the corresponding data file has been output. An optional argument is the keyword “FCT” indicating the presence of AMRFCT-type data. In this case, the file name header will change from “aIO” to “fct” and a flag will be passed on to the filtering program signifying the difference in the data format.

- Counts the number of files in the given data set and generate a list of all file names written with their absolute directory paths (the latter is necessary since Fortran doesn’t handle very well the usual Unix shortcuts).

- For each file in the list, scan the header and extract the total number of cells and the number of header lines (the latter need to be skipped by the filtering program when it reads the actual data). The total number of blocks is then computed by dividing the number of cells per block into the total number of cells.

- Save all parameters, namely the list of filenames in the data set, the number of header lines, and the total number of blocks, to a file (named “arguments”). Start the Fortran filtering module.

Since the number of blocks is now a parameter of the data structure, it is necessary to represent that data structure with arrays whose sizes can be assigned at execution time. Thus the new filtering module makes use of Fortran 90’s support for **allocatable arrays**. The module now operates as follows:

- Read the input parameters from the “arguments” file. Allocate storage space for all arrays whose size depends on the total number of blocks.

- For each file in the data set (as specified in the “arguments” file), read the data corresponding to the individual blocks comprising the solution domain.

- From the boundaries of the individual cells within each block, compute the block’s size and centroid coordinates. Along with the variables associated with each cell within a block, write these into a global data structure.

- While cycling through the files, keep track of the minimum and the maximum boundary coordinates in each dimension. Eventually, obtain the global boundary coordinates for the entire domain.
• After all input files have been read, output the global data structure to a single file that is compatible with VizFlow. The format of the VizFlow data file is as follows:

**Line 1:** The VizFlow identification string: '#VizFlow#', '#VizFlow#Spherical#', etc.

**Line 2:** Number of dimensions: data can be 2- or 3-dimensional; number of variables associated with a cell; a triplet of numbers indicating the number of cells per dimension for each block; total number of blocks

**Line 3:** Three pairs of numbers indicating the minimum and maximum values for domain coordinates in each dimension.

**Lines 4 and beyond:** Block data. Each block starts with two lines containing the block’s center-point coordinates, and the block’s size in each dimension; for each cell in the block there is a line with an n-tuple of numbers representing the variables associated with that cell.

### 2.2 The IRIS GL to OpenGL/GLUT transition

**Summary:** Our earliest version of the visualization package was based on IrisGL - a proprietary programming interface designed exclusively for Silicon Graphics machines. Since the early 1990’s IrisGL has been superseded by OpenGL, which has now become the defacto industry standard for developing graphics applications. Even though IrisGL is still supported on SGI workstations, it has very limited portability and some of its functionality is considered outdated. Thus we have converted the IrisGL portion of the code into a combination of OpenGL and GLUT. GLUT is a programming interface for writing window system independent OpenGL programs. Unlike IrisGL, OpenGL does NOT specify how rendering windows are created and manipulated, how user input events are handled etc. and so it needs a window system API that takes care of such things. GLUT provides a simple and easy to use interface that allows for rapid application development and prototyping (as opposed to native API’s such as X Window System’s Xlib, or Xt/Motif, which can be be daunting). Since GLUT expects a certain program structure that fits its own framework, the graphics portion of the visualization package had to be restructured substantially. Some parts were dropped out and others have been rewritten from scratch.

#### 2.2.1 Porting Commands with Exact Equivalents

Some IRIS GL commands have exact OpenGL equivalents and so they are a good place to start the conversion process. By “exact equivalent” we mean that the corresponding command will have the same interface, i.e. its arguments will have the same type, number etc. Actually, for IrisGL programs written in C, there is a conversion utility called **toogl** which automates this step. However, no such utility exists for Fortran programs, so command substitution even for this category of commands has to be done by hand. Below is a table of IrisGL commands used in the earliest version of the code, which have later been replaced by their exact OpenGL equivalents.

1. Note that the IrisGL commands are given according to the Fortran 77 implementation; the OpenGL commands are given according to their C implementation, but they are the same in the f90gl implementation as well (see below)
<table>
<thead>
<tr>
<th>IrisGL</th>
<th>OpenGL/GLU</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>perspe()</td>
<td>gluPerspective()</td>
<td></td>
</tr>
<tr>
<td>loadid()</td>
<td>glLoadIdentity()</td>
<td></td>
</tr>
<tr>
<td>mmode(mviewi)</td>
<td>glMatrixMode(GL_MODELVIEW)</td>
<td></td>
</tr>
<tr>
<td>mmode(mproje)</td>
<td>glMatrixMode(GL_PROJECTION)</td>
<td></td>
</tr>
<tr>
<td>pushma()</td>
<td>glPushMatrix()</td>
<td></td>
</tr>
<tr>
<td>popmat()</td>
<td>glPopMatrix()</td>
<td></td>
</tr>
<tr>
<td>pushvi()</td>
<td>glPushAttrib(GL_VIEWPORT)</td>
<td></td>
</tr>
<tr>
<td>transl()</td>
<td>glTranslatef()</td>
<td></td>
</tr>
<tr>
<td>zbuffe(.true.)</td>
<td>glEnable(GL_DEPTH_TEST)</td>
<td>The 2-dimensional versions of these commands are cmov2() and glRasterPos2f() respectively</td>
</tr>
<tr>
<td>zbuffe(.false.)</td>
<td>glDisable(GL_DEPTH_TEST)</td>
<td></td>
</tr>
<tr>
<td>lsetde()</td>
<td>glDepthRange()</td>
<td></td>
</tr>
<tr>
<td>linewi()</td>
<td>glLineWidth()</td>
<td></td>
</tr>
<tr>
<td>bgnlin()</td>
<td>glBegin(GL_LINE_STRIP)</td>
<td></td>
</tr>
<tr>
<td>endlin()</td>
<td>glEnd()</td>
<td></td>
</tr>
<tr>
<td>bgncl()</td>
<td>glBegin(GL_LINE_LOOP)</td>
<td></td>
</tr>
<tr>
<td>endcl()</td>
<td>glEnd()</td>
<td></td>
</tr>
<tr>
<td>draw()</td>
<td>glBegin(GL_LINES)</td>
<td></td>
</tr>
<tr>
<td>bgnupol()</td>
<td>glBegin(GL_POLYGON)</td>
<td></td>
</tr>
<tr>
<td>cmov()</td>
<td>glRasterPos3f()</td>
<td></td>
</tr>
<tr>
<td>endpol()</td>
<td>glEnd()</td>
<td></td>
</tr>
<tr>
<td>finish()</td>
<td>glFinish()</td>
<td></td>
</tr>
<tr>
<td>gflush()</td>
<td>glFlush()</td>
<td></td>
</tr>
<tr>
<td>ortho()</td>
<td>glOrtho()</td>
<td></td>
</tr>
<tr>
<td>multis()</td>
<td>glEnable(GL_MULTISAMPLE_SGIS)</td>
<td></td>
</tr>
<tr>
<td>lmbind()</td>
<td>glEnable(GL_LIGHTING)</td>
<td></td>
</tr>
<tr>
<td>lsetde()</td>
<td>glDepthRange()</td>
<td></td>
</tr>
<tr>
<td>rgbcol()</td>
<td>glColor3ub()</td>
<td></td>
</tr>
<tr>
<td>v3f()</td>
<td>glVertex3fv()</td>
<td></td>
</tr>
</tbody>
</table>

2.2.2 Porting Commands Whose Equivalents Are Not Exact

This category contains mainly commands whose OpenGL equivalent requires changes (syntactic and/or semantic) in the argument list. A typical example is the `lookat()` command. Its
Fortran 77 specification is as follows:

```fortran
subroutine lookat(vx, vy, vz, px, py, pz, twist)
real vx, vy, vz, px, py, pz
integer*4 twist
```

This command defines the viewpoint with coordinates \((vx, vy, vz)\) and a reference point with coordinates \((px, py, pz)\). The two points define the line of sight and the \texttt{twist} parameter measures the right-hand rotation angle of the camera about that line. The corresponding OpenGL routine is \texttt{gluLookat()} whose specification is as follows:

```fortran
subroutine gluLookAt(vx, vy, vz, px, py, pz, upx, upy, upz)
real vx, vy, vz, px, py, pz, upx, upy, upz
```

Again \((vx, vy, vz)\) and \((px, py, pz)\) represent the viewpoint and the reference point respectively, but now the vector \((upx, upy, upz)\) represents the orientation of the camera relative to the line of sight. To convert from \texttt{lookat(vx, vy, vz, px, py, pz, twist)} to \texttt{gluLookAt(vx, vy, vz, px, py, pz, upx, upy, upz)} one can, for example, use the following transformation:

\[
upx = \sin(twist \ast \pi/180) \quad upy = \cos(twist \ast \pi/180) \quad upz = 0
\]

Note that \texttt{twist} is measured in tenths of a degree, so in practice one has to divide by 1800 instead of 180 in order to convert \texttt{twist} to radians.

Below is a list of IrisGL commands whose equivalents are not exact, together with the corresponding OpenGL/glu commands and the necessary transition tips:
<table>
<thead>
<tr>
<th>IrisGL</th>
<th>OpenGL/GLU</th>
<th>Transition Tips</th>
</tr>
</thead>
<tbody>
<tr>
<td>rot(angle, axis)</td>
<td>glRotate(angle, rx,ry,rz)</td>
<td><strong>axis</strong> is one of 'x', 'y', or 'z' coordinate axis, whereas (rx, ry, rz) represent an arbitrary axis of rotation.</td>
</tr>
<tr>
<td>c3f(cv)</td>
<td>glColor3ubv(cv)</td>
<td>The color vector cv contains the red, green, and blue components of the current color given as integers between 0 and 255. This translates into a vector of type GL_UNSIGNED_BYTE in OpenGL. However, since Fortran lacks such integer type, color components need to be specially manipulated (see discussion under F90).</td>
</tr>
<tr>
<td>czclear(cv, zv)</td>
<td>glClearIndex(cv); glClearDepth(zv)</td>
<td>Sets the color and the depth bitplanes in the current framebuffer to the values specified by cv and zv. The corresponding OpenGL command assumes that the values of the respective bitplanes have been previously prescribed by the commands glClearColor and glClearDepth.</td>
</tr>
<tr>
<td>viewpo(l,r,b,t)</td>
<td>glViewPort(x,y,w,h)</td>
<td>The viewport in IrisGL is specified by the (window) coordinates of the left, right, bottom, and top side of the viewing rectangle. In OpenGL it is specified by the coordinates (x, y) of the lower left corner plus the width and the height (w, h) of the viewing rectangle.</td>
</tr>
<tr>
<td>getvie(l,r,b,t)</td>
<td>glGetInteger(GL_VIEWPORT, param)</td>
<td>getvie returns the coordinates of the left, right, bottom, and top side of the current viewport. The corresponding OpenGL command returns the coordinates of the viewport’s lower left corner plus its width and height (as elements of the array param). See the notes on glViewPort.</td>
</tr>
<tr>
<td>rot(angle, axis)</td>
<td>glRotatef(angle/10, x,y,z)</td>
<td>The <strong>angle</strong> parameter in <strong>rot</strong> is measured in tenths of a degree, while <strong>glRotatef</strong> expects an angle in degrees. The <strong>axis</strong> parameter in <strong>rot</strong> is a literal - ’x’, ’y’, or ’z’ - indicating the axis of rotation, while <strong>glRotatef</strong> accepts an arbitrary axis defined by the coordinates of the vector (x, y, z).</td>
</tr>
</tbody>
</table>
2.2.3 Redundant IrisGL Commands

Some IrisGL commands are either no longer supported, or their functionality is redundant in the context of OpenGL/GLUT. Such commands can safely be eliminated from the code. Examples are:

<table>
<thead>
<tr>
<th>Command</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>reshap</td>
<td>no longer supported</td>
</tr>
<tr>
<td>foregr</td>
<td>keeps a process in the fg: always true in OpenGL</td>
</tr>
<tr>
<td>subpix</td>
<td>controls the placement of points, lines, etc.; in OpenGL, subpixel mode is always on.</td>
</tr>
<tr>
<td>gconfi</td>
<td>sets the requested modes of the current window (see GLUT)</td>
</tr>
<tr>
<td>lsetde</td>
<td>sets the range of the depth buffer; can be replaced with glDepthRange, but is not necessary; in OpenGL default range is [0.0, 1.0]</td>
</tr>
<tr>
<td>getbut</td>
<td>tests the state of a (mouse) button (see GLUT) use GLUT mouse callback function instead (see below).</td>
</tr>
<tr>
<td>getval</td>
<td>returns the value stored in a given device (see GLUT)</td>
</tr>
<tr>
<td>mssize</td>
<td>specifies multisample buffer configuration (see GLUT)</td>
</tr>
<tr>
<td>zbsize</td>
<td>specifies the size of the main z-buffer (in number of bit-planes); when multisampling is used, this is usually set to 0, since the main z-buffer takes resources away from the multisampling buffers; no longer needed since GLUT takes care of this when multisampling buffer is requested (see GLUT notes)</td>
</tr>
</tbody>
</table>

2.2.4 Porting the Windowing Interface

As mentioned earlier, OpenGL is *window system independent*. This means that the basic OpenGL routines do not describe how an OpenGL context is created or how to create a window for use with OpenGL rendering. Instead, OpenGL leaves the details of window creation and other window management tasks to the native window system. In contrast, IrisGL supports a subset of routines which provide high-level access to the X Window System, and at the same time hide the details of the X protocol from the graphics programmer. Several possibilities exist for handling these routines when converting IrisGL code to OpenGL:

1. Use Xlib via GLX (the OpenGL X server extension). This is not for the faint at heart! Since Xlib is the lowest-level programming interface to the X protocol, a great deal of detail needs to be taken care of by the programmer. True, this method provides the most flexibility in creating GUI functionality but it is very cumbersome and requires a large amount of code to be written just for describing a single task.

2. Use Xt (X Toolkit Intrinsics) with some widget set (OSF/Motif, Athena, etc.), along with the GLw*DrawingArea widgets. Apart from lending a consistently professional look and
feel to the GUI, these widget classes provide useful resources and callbacks to the OpenGL program. Also, Motif and Xt take care of many routine but complicated issues associated with window management and event handling that are otherwise left to a Xlib/GLX program. Yet, programming with Xt requires a great deal of experience and could still be rather cumbersome. Moreover, Motif is proprietary software and its portability is somewhat limited. Another consideration is that both the Xt/Motif and the Xlib/GLX libraries are available with C bindings only.

3. Use GLUT (Graphic Library Utility Toolkit). This public domain toolkit, developed by Mark Kilgard in the early 1990’s, is much simpler to use since it provides a higher-level interface to the windowing environment. It is freely available, highly portable (implementations exist even for Windows NT and OS/2), and allows for quick prototyping and development. True, it is not a fully featured window system API (like Xt/Motif, or Xlib/GLX for example), but it still supports basic functionality such as window management, event handling and pull-down menus. Moreover, GLUT has both C and Fortran 77/90 bindings which makes it an excellent candidate for integrating with OpenGL programs written in Fortran. For those reasons we chose GLUT as a window system API for the current version of VizFlow.

In the future, we plan to rewrite VizFlow in C/C++ and use a state-of-the-art public-domain application development framework such as Qt or GTK. Both of these allow for rapid prototyping and provide a modern multiplatform object-oriented programming environment with excellent GUI functionality. Rewriting the code in C/C++ will also have an additional benefit in terms of optimizing the visualization algorithms and data structures.

Converting the Event Processing Loop to GLUT’s callback model

Having chosen GLUT over other possible API’s means that certain changes need to be made in the program structure in order to reflect GLUT’s utilization of what is called “inverted” or callback-driven programming style. This means the GLUT program registers functions to be called by the toolkit when high-level events occur, e.g. when a window needs to be redrawn, when user input is received, etc. Since GLUT takes care of the event loop management, the portion of the IrisGL program that used to be responsible for this should be split into subroutines, which are then registered as callbacks. For example, suppose in the IrisGL code, we have the following segment:

1000 continue

call winset(window_id) !## activates window with ID=window_id
event = qtest() !## check for keyboard input events
if (event .ne. 0) then !!# if there are events, start processing them
dev = qread(val) !!# dev will indicate the type of input
!!# device, and val will be the the ASCII
!!# value of the key that has been pressed.
key = char(val) !!# converts the ASCII value to a character.
20   if (key .eq. 'a') then
!!# Go through the set of the program's control keys, check if any of them
!!# had been pressed and take corresponding actions.

200   endif

endif

goto 1000

To translate this into GLUT we need to place the portion where key values are being checked
(between lines 20 and 200) into a separate subroutine. For example:

subroutine keyboard(val, x, y)

integer val, x, y
character key

key = char(val)

if (key .eq. 'a') then

.......   
.......   

!!# Do the checks for program control keys and take corresponding actions

.......   
endif

end subroutine keyboard

This subroutine has to be in the global name space of the program so it is accessible by GLUT
at any time. We also need to put a callback registration command in the GLUT initialization
sequence indicating that the subroutine should be called whenever there is an input event from
the keyboard. In this case, the command would be:

call glutKeyBoardFunc(keyboard)

Note that the keyboard subroutine has a certain interface assumed by GLUT, namely, it has
two more input arguments besides the ASCII value of the pressed key. The pair \((x, y)\) indicates
the position of the mouse (in screen coordinates) when the given key was pressed and though
this information is not always needed, these arguments must be included in the declaration for
the callback to be considered valid. On the other hand the subroutine name could be arbitrary
as long as it matches the one declared with \texttt{glutKeyBoardFunc}.

In a similar manner, callbacks for processing mouse events, menus, window resizing, redisplay
etc. can be set up. All such callback functions have to be registered prior to entering GLUT’s
main loop. A sample GLUT initialization sequence is shown below:
call glutInit
call glutInitDisplayMode(GLUT_RGBA + GLUT_DOUBLE + GLUT_DEPTH)
call glutInitWindowSize(width, height)
call glutInitWindowPosition(WinPos_x, WinPos_y)
iw = glutCreateWindow(name)
call glutKeyboardFunc(keyboard)
call glutDisplayFunc(display)
call glutMotionFunc(motion)
call glutMouseFunc(mouse)
call glutSpecialFunc(special)
call glutMainLoop

This sequence initializes a window of specified width and height, positioned at (WinPos_x, WinPos_y), with a given name, and requests a double framebuffer with both RGBA-color and depth components to be associated to that window. After that several callback functions are registered and finally the program enters the GLUT main loop. At this point all control is passed on to GLUT, which will invoke registered callbacks whenever necessary.

**Porting Specific Windowing Commands**

Certain IrisGL windowing commands have direct (or “semi-direct”) GLUT counterparts. A list of such commands used in the original version of VizFlow is given below:
<table>
<thead>
<tr>
<th>IrisGL</th>
<th>GLUT</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>glutInitDisplayMode</td>
<td>GLUT argument is GLUT_DOUBLE</td>
</tr>
<tr>
<td>cmode</td>
<td>glutInitDisplayMode</td>
<td>GLUT argument is GLUT_RGBA</td>
</tr>
<tr>
<td>swapbu</td>
<td>glutSwapBuffers</td>
<td></td>
</tr>
<tr>
<td>winset</td>
<td>glutSetWindow</td>
<td></td>
</tr>
<tr>
<td>winpos</td>
<td>glutInitWindowPosition</td>
<td>empty the event queue; for this and other event queue manipulation commands use GLUT callback functions instead</td>
</tr>
<tr>
<td>qreset</td>
<td>redundant</td>
<td></td>
</tr>
<tr>
<td>qdevic</td>
<td>redundant</td>
<td>registers a device in the event queue</td>
</tr>
<tr>
<td>qread</td>
<td>redundant</td>
<td>reads the first event in the event queue</td>
</tr>
<tr>
<td>qreset</td>
<td>redundant</td>
<td>empties the event queue</td>
</tr>
<tr>
<td>gconfi</td>
<td>redundant</td>
<td>sets the requested modes of the current window</td>
</tr>
<tr>
<td>getbut</td>
<td>redundant</td>
<td>tests the state of a (mouse) button</td>
</tr>
<tr>
<td>getval</td>
<td>redundant</td>
<td>returns the value stored in a given device</td>
</tr>
<tr>
<td>wincon</td>
<td>redundant</td>
<td>binds the window constraints (size, position, etc.) to the current graphics window</td>
</tr>
<tr>
<td>winope</td>
<td>glutCreateWindow</td>
<td></td>
</tr>
<tr>
<td>prefpo (l,r,b,t)</td>
<td></td>
<td>w = r – l and h = t – b specify, respectively, the width and the height of the current window (in pixels).</td>
</tr>
<tr>
<td>mssize</td>
<td>redundant</td>
<td>set multisampling buffers parameters; in GLUT, it suffices to request a multisampling buffer with glutInitDisplayMode(GLUT_MULTISAMPLING) and enable it with glEnable(GL_MULTISAMPLE_SGIS); note that multisampling is supported only on SGI Reality Engine machines.</td>
</tr>
<tr>
<td>zbsize</td>
<td>redundant</td>
<td></td>
</tr>
<tr>
<td>subpix</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Porting Font Management Calls**

In IrisGL text can be rendered either using intrinsic commands, such as charst, defrasterfont, textlfont etc., or via the IRIS font manager utility routines (accessible from Fortran only via C-wrappers). Both approaches have been utilized in the initial VizFlow version. Since OpenGL has no explicit routines for rendering text, the first approach becomes obsolete. The second one can still be pursued, however it entails an overhead not only from accessing the IRIS Font Manager but also from doing it via a wrapper program.

A viable alternative is to use the GLUT library font rendering routines: glutBitmapCharacter, or glutStrokeCharacter. We have chosen the former, since it corresponds almost directly to IrisGL’s charstr except that it renders one character at a time, as opposed to an arbitrary string of text. The following segment illustrates how a single call, say charst(string, 10), can be converted to OpenGL (in this case string is a string array of length 10):

```fortran
do i=1,len(string)
```
call glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, &
   IACChar(string(i:i)))
end do

The first argument of \texttt{glutBitmapCharacter} specifies the font to be used for rendering and the second is a character in its ASCII representation. Note that Fortran requires explicit character-to-integer conversion, as opposed to C, where a character and its ASCII representation are indistinguishable. Several bitmap fonts are available via this routine: Times-Roman (12pt, or 24pt), Helvetica (10pt, 12pt, or 18pt), and fixed-width (8x13, or 9x15).

2.3 Fortran 77 to Fortran 90 transition

\textbf{Summary:} Many reasons necessitated the porting the code to Fortran 90, among which were: support for allocatable arrays, data encapsulation and better program organization via modules, arbitrary name lengths, whole-array operations, advanced data types etc. Since OpenGL’s bindings for Fortran 77 and Fortran 90 use different name conventions, however, it was much more reasonable to carry out the transition from IrisGL to OpenGL/GLUT, and from Fortran 77 to Fortran 90, simultaneously, in order to avoid changing all OpenGL/GLUT command names twice. This presented itself as a challenge in terms of organization and planning since no testing could be done until the entire transition was over, so extra care needed to be taken about preserving the integrity of the program’s algorithms and data structures.

2.3.1 Fortran 90 implementations of OpenGL/GLUT

Even though the Fortran 90 standard for OpenGL has been designed more than five years ago, there is only one implementation so far, namely the one developed by William Mitchell at the Information Technology Lab at NIST, called “F90GL” [A Fortran Interface for OpenGL, William Mitchell, NISTIR 6134, National Institute of Standards and Technology]. It is available in the public domain, and can be ported virtually on any OpenGL-ready platform equipped with a Fortran compiler. The method used for creating Fortran 90 bindings for the OpenGL routines is via C-wrappers that provide interface to the original C libraries. Even though this adds a slight overhead to all OpenGL calls, it is not very significant and in most cases can be minimized by using precompiled display lists of geometric primitives. A major advantage of the new Fortran 90 interface is that, and unlike the existing Fortran 77 implementations, it provides access to the full functionality of OpenGL, and does not depend on any extensions to the Fortran standard. This enhances the portability and the robustness of the user application code, and increases the similarity between the Fortran and C interfaces.

F90GL can be downloaded from William Mitchell’s web site at \url{http://math.nist.gov/f90gl}. Before installing the package, one has to download and install the latest version of GLUT (version 7.3 of November 1998), which contains a compatible Fortran 90 implementation of the GLUT library (available at: \url{http://reality.sgi.com/mjk_asd/glut3/}). The result of installing these two packages is a set of libraries: \texttt{libf90GL}, \texttt{libf90GLU}, and \texttt{libf90glut}, plus a set of Fortran 90 modules, which provide programming interface to the graphics libraries, namely: \texttt{OPENGL.GL}, \texttt{OPENGL.GLU}, and \texttt{OPENGL.GLUT}. To access any OpenGL/GLUT function, a Fortran 90 program unit must have these statements in its header:
use OPENGL_GL
use OPENGL_GLU
use OPENGL_GLUT

Then for compilation, the following options need to be included in the compiler’s command line: -lf90glut -lf90GLU -lf90GL, before the usual -lglut -lGLU -lGL.

2.3.2 Caveats

Even though the Fortran 90 and the C bindings are very similar, there are still some caveats, mostly due to some data representation differences in the two languages. In general, data type correspondence is achieved through use of KIND type parameters on the Fortran side. For example, GLfloat corresponds to real (kind = GLfloat), and GLshort - to integer (kind = GLshort). The wrapper interface tries to match the C representation of an entity of a given OpenGL type to the Fortran representation of that entity whenever possible. Results may vary depending on the compiler, hardware platform etc. but in any case, the lack of a particular Fortran representation remains transparent to the user.

Using the GLubyte type

Parameters of type GLubyte are needed for example in color-setting routines such as glColor3ub, and glColor3ubv. In the initial VizFlow version, the color map used to represent cell-data values is given in terms of RGB unsigned-byte-type integers, i.e. triples of numbers in the range [0, 255]. Fortran, however, does not support unsigned integer types; signed integers of the same size are used instead. For example, if red is declared as integer (GLubyte), the assignment red = 128, will result in ‘constant out of bounds’ error since the corresponding Fortran type - signed byte - has its range [−128, 127]. Several approaches can be taken to avoid this problem. One is to assign the value 128 to some temporary variable of “larger” integer type and then copy its value into the unsigned-byte-type variable. For example suppose x is of type GLint; then the assignments:

\[
x = 128
\]
\[
red = x
\]

will initialize red with the correct value. Note that even though in this case Fortran will think of red as −128, the OpenGL interface will interpret red in its proper context, i.e. as an unsigned byte with hexadecimal value 80.

Another approach is to assign to red either the appropriate signed value, or use Fortran’s intrinsic function IBSET with the appropriate unsigned valueto set the leading bit to 1. This means that in the first case we should set red = −127, and in the second: red = IBSET(7, 0).

Note that no conversion is necessary if unsigned-byte-type data is read from a file with default input control. For example the command: read (file,*) red will obtain the value 128 from file and then copy it to red. Just as in the example above where a temporary variable was used, red will end up with hexadecimal value 80, which will be interpreted by Fortran as −128, but by OpenGL as 128.
## Constants

In order to ensure proper type assignment, the Fortran interface requires that constants have the kind type parameter attached, e.g. 1.0_GLfloat. This is mandatory for constants that are passed as actual arguments to OpenGL functions; otherwise the Fortran compiler will not be able to recognize the function’s interface and will fail. For example, in:

```fortran
  call glVertex3f(1.0_GLfloat, 1.0_GLfloat, 0.0_GLfloat);
  call glVertex3f(1.0, 1.0, 0.0)
```

the first call to glVertex is valid and the second one is not. Attaching the type kind parameter to constants may get rather tedious, so as an alternative, one may instead assign the constants to variables of the same kind type and then pass the variables as actual arguments. For example, if x, y, and z are declared as `real (GLfloat)`, then:

```fortran
  x=1.0; y=1.0; z=0.0
  glVertex3f(x, y, z)
```

will be yield the proper result.

## Arrays as actual arguments

Some OpenGL functions take arguments of array type, which in the C implementation are passed as pointers to appropriate array elements. For example, the C specifications:

```c
void glVertex3fv( const GLfloat *v )
```

means that in C, the following will be valid calls to glVertex3fv:

```fortran
  float single_vertex[3];
  float vertex_array[10,3];
  .......
  glVertex3fv(single_vertex);
  glVertex3fv(&vertex_array[0,0]);
  glVertex3fv(&vertex_array[5,0]);
```

In the first instance, single_vertex is a one-dimensional array, so single_vertex and &single_vertex[0] are equivalent. In the next two instances, a pointer to the 0-th and the 5-th row, respectively, of vertex_array designates a vector of size 3, which is a valid argument of glVertex3fv.

In order to conform to the C specification, a Fortran call to an OpenGL function with array type arguments needs to obtain a pointer to the appopriate array element. In the case of a 1-dimensional array - just as in the C case - the array name is the same as a pointer to the first element. In the multidimensional case, a pointer can be constructed using the array section operator. For example:

```fortran
  real (GLfloat) :: single_vertex(3)
  real (GLfloat) :: vertex_array(3,10);
  .......
  call glVertex3fv(single_vertex);
  call glVertex3fv(vertex_array(1:,1));
  call glVertex3fv(vertex_array(1:,5));
```
will correspond to the C example above. Note that in Fortran, arrays are stored in column-major format, and the index of the first element in each column starts (by default) from 1.

Structures

Some OpenGL-related libraries (in our case GLU, the OpenGL Utility Library) define structures that are used as procedure arguments. In the Fortran interface these structures correspond to derived types. For example the GLU type `gluQuadricObj`, used for constructing quadric surfaces, can be declared as `TYPE(gluQuadricObj)` in Fortran.

Pointers

Some OpenGL and OpenGL-related procedures take pointers as actual arguments, or provide pointers as return values. For example the function `gluNewQuadric` initializes a new quadric object and returns a pointer to it. The C specification of this function is:

```c
GLUquadric* gluNewQuadric( void )
```

In order to use `gluNewQuadric` an object of type pointer to `gluQuadricObj` needs to be declared. In Fortran, this can be done in the following way:

```fortran
TYPE(gluQuadricObj), pointer :: quadsurf
........
quadsurf = gluNewQuadric()
```

Another use of pointers is when passing function names as arguments to certain procedures. Such situation arises, for example, when a GLUT callback function has to be temporarily disabled. If we would like to suspend, say input from the keyboard, we would normally, in a C program that is, pass a “null” pointer to `glutKeyboardFunc`. Since Fortran does not support such a pointer type, the interface provides a derived type constant, called `glutnullfunc`, that serves this purpose. So the proper way to disable the keyboard input would be to use: `glutKeyboardFunc(glutnullfunc).`
Chapter 3

Functional Development

3.1 Spherical Data

A major VizFlow functionality is the capability to visualize data given in spherical coordinates. Data sets containing “spherical data” follow the same file format as Cartesian ones, except that a file header will start with the string '#VizFlow#Spherical#' to indicate the respective data type. Domain coordinates are interpreted as triples of numbers \((r, \theta, \phi)\), where \(r\) is the polar radius, while \(\theta\) and \(\phi\) are the latitudinal and longitudinal angles respectively. Both angles are measured in degrees: \(\theta\) from the positive \(z\)-axis, and \(\phi\) counter-clockwise from the positive \(x\)-axis. A domain given in spherical coordinates as \([r_{\text{min}}, r_{\text{max}}] \times [\phi_{\text{min}}, \phi_{\text{max}}] \times [\theta_{\text{min}}, \theta_{\text{max}}]\) looks like a spherical “wedge”. Just like with Cartesian domains, we need to visualize domain cross-sections, i.e. level sets of the coordinate functions \(r, \theta, \phi\). Unlike the Cartesian case, however, where cross-sections are all geometrically the same, namely rectangular polygons, in the spherical case each coordinate level set is different. The level sets \(\rho = \text{const.}\) are spherical polygons of the form: \([\theta_1, \theta_2] \times [\phi_1, \phi_2]\); the level sets \(\theta = \text{const.}\) are conical polygons of the form: \([\rho_1, \rho_2] \times [\phi_1, \phi_2]\), whereas the level sets \(\phi = \text{const.}\) are planar annular polygons of the form: \([\rho_1, \rho_2] \times [\theta_1, \theta_2]\) (see Fig. 3.1).

Unfortunately, OpenGL has direct support for graphics primitives like these only in the case of annular regions (the \texttt{gluDisk} command). In order to render spherical or conical polygons, we use a representation of such polygons as Non-uniform Rational B-spline (NURBS) surface patches and OpenGL’s NURBS interface. Below is a discussion of the theoretical and practical aspects of the NURBS rendering algorithm.

3.1.1 NURBS Basics

NURBS curves and surfaces are ubiquitous in Computer-Aided Modeling and Design. Their theory and practice, however, is a vast subject so we present here only a brief summary of features that we use in our particular implementation. For a comprehensive introduction, see for instance Farin [5], or The NURBS Book, [12].
Figure 3.1: Level sets of the spherical coordinate functions

**NURBS Curves**

One of the most common representations of curves and surfaces is in terms of parametric equations. In the plane, for example, a curve can be given as a pair of functions (or equivalently, a vector-valued function) of an independent parameter:

\[ \mathbf{C}(t) = (x(t), y(t)) \quad a \leq t \leq b \]

For instance, the first quadrant arc of a circle of radius 1, centered at the origin, can be defined by parametric equations as:

\[ x(t) = \cos(t) \]
\[ y(t) = \sin(t) \quad 0 \leq t \leq \frac{\pi}{2} \]

Setting \( u = \tan(t/2) \) gives another representation of the same arc:

\[ x(u) = \frac{1 - u^2}{1 + u^2} \]
\[ y(u) = \frac{2u}{1 + u^2} \quad 0 \leq u \leq 1 \]

Thus, the parametric representation of a curve is not unique. If a curve can be represented in a variety of different ways, it is natural to ask: which ones are the best? In other words, what category of functions is the most suitable for our situation? For computer based applications this is clearly the category of rational functions, i.e. functions that can be written as quotients of polynomials, such as in equation (3.3). Polynomials are superior in terms of computational efficiency, numerical precision, and storage requirements,
and although not every curve possesses a rational representation, the ones that do, comprise a fairly large class that contains most of the commonly known curves.

Let \( p(u) \) be a real \( n \)-th degree polynomial on one variable, defined over a closed interval \([a, b]\). Without loss of generality, we may assume that \( a = 0 \) and \( b = 1 \). Instead of writing \( p \) with respect to the usual monomial basis \( \{u^i\}_{i=0}^{n} \), we can write it terms of the so called Bernstein-Bézier basis, given by the polynomials:

\[
B_{i,n}(u) = \binom{n}{i} u^i (1-u)^{n-i}, \quad 0 \leq u \leq 1
\]

In other words, over the interval \([0,1]\):

\[
p(t) = \sum_{i=0}^{n} P_i B_{i,n}(u)
\]

If the coefficients \( P_i \) are vectors of rank \( k \), then \( p(t) \in \mathbb{R}^k \), so in effect \( p \) defines a map from \([0,1]\) into \( \mathbb{R}^k \), i.e. a \( k \)-dimensional curve segment. A number of nice properties make the Bernstein-Bézier basis very desirable for computational modeling of curves; for example, if we interpret the \( P_i \)'s as points in \( \mathbb{R}^2 \), then \( p(u) \) will be in their convex hull for any \( 0 \leq u \leq 1 \). The set of points \( \{P_i\}, i = 0 \ldots n \), forms what is known as control polygon.

More generally, a rational Bézier curve of degree \( n \) is defined as:

\[
C(u) = \frac{\sum_{i=0}^{n} w_i B_{i,n}(u) P_i}{\sum_{i=0}^{n} w_i B_{i,n}(u)}
\]

The scalars \( \{w_i\}, i = 0 \ldots n \), are called weights, and although their meaning is not immediately obvious from this context, it suffices to say that they introduce additional degrees of freedom in determining the shape of a curve. In particular, since \( \sum_{i=0}^{n} B_{i,n}(u) = 1 \), if all \( w_i \)'s are equal to \( 1 \), then \( C(u) \) reduces to a polynomial curve.

For example, the first-quadrant circular arc mentioned before, has the following rational Bézier representation (see Fig. 3.2)

\[
C(u) = \frac{(1-u)^2 w_0 P_0 + 2u(1-u)w_1 P_1 + u^2 w_2 P_2}{(1-u)^2 w_0 + 2u(1-u)w_1 + u^2 w_2}, \quad u \in [0,1]
\]

with control points: \( P_0 = (0, 1) \), \( P_1 = (1, 1) \), \( P_2 = (1, 0) \), and weights: \( w_0 = w_2 = 1 \), \( w_1 = \sqrt{2}/2 \).

The class of rational curves represented in this way can be enlarged even further if as a basis we consider a set of \textit{piecewise polynomial} functions on an interval \([a, b]\) that satisfies properties similar to those of \( \{B_{i,n}(u)\} \). We start with a set of parameter values \( U = \{u_0, u_1, \ldots, u_l\} \), called knots, which form a partition of the interval \([a, b]\), such that \( a = u_0 \leq u_1 \leq \cdots < u_l = b \). The set \( U \) is called a knot vector. The \( i \)-th B-spline basis function of degree \( p \) (order \( p+1 \)), denoted by \( N_{i,p}(u) \) is defined as:

\[
N_{i,0}(u) = \begin{cases} 
1 & \text{if } u_i \leq u < u_{i+1} \\
0 & \text{otherwise}
\end{cases}
\]

\[
N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u)
\]

\[25\]
Thus $N_{i,0}(u)$ is a step function, and for $p > 0$, $N_{i,p}(u)$ is a linear combination of two $(p - 1)$-degree polynomials. Now, for a given knot vector $U = \{0, \ldots, 0, u_{p+1}, \ldots, u_{n-p-1}, 1, \ldots, 1\}$, a $p$-th degree rational B-spline curve is defined by:

\begin{equation}
C(u) = \frac{\sum_{i=0}^{n} w_i N_{i,p}(u) P_i}{\sum_{i=0}^{n} w_i N_{i,p}(u)}
\end{equation}

The repeated knot values are required in order to guarantee certain differentiability conditions at the end points. Using B-spline rational parametrization allows for even more flexible representation of curves. For example, it is well known that there is no second-degree rational Bézier parametrization of a full circular arc. Various B-spline parametrizations exist however; for instance there is one that uses a nine-point square control polygon, a knot vector $U = \{0, 0, 0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1, 1, 1\}$, and weights: $\{w_i\} = \{1, \frac{\sqrt{2}}{2}, 1, \frac{\sqrt{2}}{2}, 1, \frac{\sqrt{2}}{2}, 1, \frac{\sqrt{2}}{2}, 1\}$. The control points are at the vertices and at the midpoints of a square circumscribing the circle; for a circle of radius 1, centered at the origin, for example, these would be the points:

\[\{P_i\} = \{(1, 0), (1, 1), (0, 0), (-1, 1), (-1, 0), (-1, -1), (0, -1), (1, -1), (1, 0)\}\]

Notice that over each subinterval defined by adjacent knots, $N_{i,2}(u)$ is identical with the restriction of the Bernstein-Bézier polynomial $B_{i,2}(u)$ to that subinterval. Thus, when $0 \leq u \leq \frac{1}{4}$, the point $C(u)$ traces a $90^\circ$-arc in the first quadrant; when $\frac{1}{4} \leq u \leq \frac{1}{2}$ - it traces a $90^\circ$-arc in the second quadrant, and so forth. Geometrically, this means that four $90^\circ$-arcs - one in each
quadrant - given with their rational Bézier parameterization - have been glued at the end-points to make a full circle defined by a single equation:

\[ C(u) = \frac{\sum_{i=0}^{8} w_i N_{i,2}(u) P_i}{\sum_{i=0}^{8} w_i N_{i,2}(u)} \]

More generally, if the knot vector of a \( p \)-degree rational B-spline curve is:

\[ U = \{0, \ldots, 0, 1, \ldots, 1\} \]

then its NURBS parametrization reduces to a Bernstein-Bézier’s.

NURBS Surfaces

The idea behind NURBS surfaces can be carried over verbatim from the discussion about NURBS curves. By definition a parametric surface patch is a map from \( D = [a, b] \times [c, d] \) to \( \mathbb{R}^k \), so in particular when \( k = 3 \), it can be given as a vector-valued function on two parameters:

\[ S(u, v) = (x(u, v), y(u, v), z(u, v)), \quad a \leq u \leq b \text{ and } c \leq v \leq d \]

If the coordinate functions \( x(u, v) \), \( y(u, v) \), and \( z(u, v) \) are restricted to the category of piecewise-rational functions on \( D \), then \( S(u, v) \) defines a NURBS surface patch over \( D \). More precisely, let \( U = \{0, \ldots, 0, u_{p+1}, \ldots, u_{r-p-1}, 1, \ldots, 1\} \) and \( V = \{0, \ldots, 0, v_{q+1}, \ldots, v_{s-q-1}, 1, \ldots, 1\} \), be knot vectors on the interval \([0, 1]\). Then a NURBS surface patch of degree \( p \) in the \( u \) direction and degree \( q \) in the \( v \) direction is the function:

\[ S(u, v) = \frac{\sum_{i=0}^{n} \sum_{j=0}^{m} w_{ij} N_{i,p}(u) N_{j,q}(v) P_{ij}}{\sum_{i=0}^{n} \sum_{j=0}^{m} w_{ij} N_{i,p}(u) N_{j,q}(v)} \quad 0 \leq u, v \leq 1 \]

where \( n = r - p - 1 \), and \( m = s - q - 1 \). The points \( \{P_{ij}\} \) form a bi-directional control mesh and the \( \{w_{ij}\} \) are the corresponding weights. It becomes clear from the formula that the surface patch \( S(u, v) \) is a tensor product of two transverse families of NURBS curves, defined respectively on the columns and on the rows of the control mesh \( P_{ij} \) and the weights matrix \( w_{ij} \).

3.1.2 Modeling Curvilinear Polygons with NURBS

We return now to the the problem of modeling spherical domain cross-sections. Recall that in this case there are three different types of curvilinear polygons: spherical, conical, and annular. Our goal is the following: given a curvilinear polygon in terms of spherical coordinates, find a control mesh \( P_{ij} \), a weights matrix \( w_{ij} \), and knot vectors \( U \) and \( V \), that define a NURBS representation of that polygon.

We start with an important observation: each of the three types of polygons that we just
mentioned, is in fact a patch on a surface of revolution, i.e. it is generated by rotating a planar curve segment through some angle around a fixed axis in 3-space. The curve segment is either a circular arc yielding a spherical polygon, or a straight line interval yielding either a conical or an annular polygon, depending on the axis of rotation. Thus, we can build a single interface that handles all three cases with the same algorithm, changing only the generating curve, the angle, and the axis of rotation.

We first present a brief outline of such an algorithm, and then proceed with a more detailed description. The gist of the algorithm is based on the fact that the a NURBS surface-of-revolution patch is essentially a tensor product of its generating curve and a circular arc of a given angle.

**Input:** A NURBS curve in 3-space, defined by a set of control points \( \{ P_j \}_{j=0}^m \), weights \( \{ w_j \}_{j=0}^m \), and a knot vector \( V \); an axis of rotation \( L \), specified by a point \( p \), and a unit vector \( u \); an angle of rotation \( \theta \).

**Output:** A NURBS surface patch, given as a set of control points \( \{ Q_{ij} \}_{i=0}^n, j=0\ldots m \), weights \( v_{ij} \), and knot vectors \( U_1 \) and \( U_2 \).

1. compute the weights \( \{ \overline{w}_i \}_{i=0}^n \), and the knot vector \( U_1 \) for a circular arc of angle \( \theta \).
2. for \( j = 0 \ldots m \)
3. find the plane \( \pi \) through \( P_j \) that is perpendicular to \( L \).
4. find the intersection point \( O = L \cap \pi \) and the distance \( r = |\overrightarrow{OP_j}| \).
5. define a local orthonormal frame \( \{ O, e_1, e_2 \} \) in \( \pi \), such that \( e_1 \) is the unit vector along \( \overrightarrow{OP_j} \), and \( e_2 \) complements \( \{ e_1, u \} \) to a right-oriented frame in \( \mathbb{R}^3 \).
6. set \( Q_{0j} = P_j \) and \( v_{0j} = w_j \)
7. generate the control polygon \( Q_{ij} \) for a circular arc of radius \( r \) and angle \( \theta \) with respect to the local frame in \( \pi \), starting at \( Q_{0j} \).
8. for \( i = 1 \ldots n \), set \( v_{ij} = \overline{w}_i w_j \)
9. go to 2
10. set \( U_2 = V \)

Note that the weights \( \overline{w}_i \) and the knot vector \( U_1 \) depend only on \( \theta \), so they can be computed before the control mesh points.

An essential ingredient in this algorithm is the generation of a NURBS circular arc and this is what we discuss next.
NURBS Circular Arcs

Earlier, we presented NURBS parametrizations of a 90°-arc and a full circle in the plane. In this section we will describe an algorithm for constructing a NURBS circular arc in arbitrary position in 3-space, given a local vector frame, an arc angle, and a radius.

We begin with the planar case, assuming that the circle is centered at the origin and defined by a radius \( r \), and initial and terminal angles \( \theta_s \) and \( \theta_e \) respectively. In other words, its trigonometric parametrization is:

\[
C(u) = (r \cos u, r \sin u), \quad \theta_s \leq u \leq \theta_e
\]

Assume for now that \( \Delta \theta = \theta_e - \theta_s \leq 90° \). Then the control polygon for the NURBS representation of the arc consists of the arc’s end points \( P_0 \) and \( P_2 \) plus the intersection point of the tangent lines \( T_0 \) and \( T_2 \) at \( P_0 \) and \( P_2 \) respectively. In particular:

\[
P_0 = (r \cos \theta_s, r \sin \theta_s) \quad (3.8)
\]

\[
P_1 = T_0 \cap T_2
\]

\[
P_2 = (r \cos \theta_e, r \sin \theta_e)
\]

The weights associated with the vertices of the control polygon are:

\[
w_0 = w_2 = 1, \quad w_1 = \cos\left(\frac{\Delta \theta}{2}\right)
\]

and since the arc is less than 90°, it affords the rational Bernstein-Bézier parametrization given by formula (3.5). Equivalently, we can set the knot vector to \( U = \{0, 0, 0, 1, 1, 1\} \) and use formula (3.7) with \( p = 2 \) and \( n = 2 \).

When the circular arc is in arbitrary position in 3-space, more initial data is needed in addition to \( r, \theta_s, \) and \( \theta_e \). More precisely, suppose that the following is given:

- \( O \): center of circle, which is also the origin of the local coordinate system
- \( \{X, Y\} \): orthonormal frame in the plane of the circle
- \( r \): radius
- \( \theta_s, \theta_e \): initial and terminal angles measured with respect to \( X \)

Hence, in terms of the familiar trigonometric parametrization, the arc is defined by:

\[
C(u) = O + r \cos uX + r \sin uY, \quad \theta_s \leq u \leq \theta_e
\]

With respect to the local coordinate system, the control polygon for the NURBS representation is given by formula (3.8) and hence we can write it in global coordinates as:

\[
P_0 = O + r \cos \theta_s X + r \sin \theta_s Y
\]

\[
P_1 = T_0 \cap T_2
\]

\[
P_2 = O + r \cos \theta_e X + r \sin \theta_e Y
\]

The weights associated with the vertices of the control polygon are the same as before:

\[
w_0 = w_2 = 1, \quad w_1 = \cos\left(\frac{\Delta \theta}{2}\right)
\]

Finding the intersection point of the tangent lines \( T_0 \) and \( T_2 \) directly, in the 3D-case, may not be quite straightforward due to numerical imprecision, so we use an alternative method that
avoids such a problem. In particular, we notice that $P_1$ lies on the bisectrix of angle $\angle \overrightarrow{P_0O}\overrightarrow{P_2}$ and therefore $\overrightarrow{OP_1}$ is proportional to the unit vector $u$ along $(\overrightarrow{OP_0} + \overrightarrow{OP_2})$. It is not hard to see that, in fact $\overrightarrow{OP_1} = rw_1 u$.

We now consider the case when $\Delta \theta > 90^\circ$. The idea is to split the arc into a number of smaller arcs, each of length less than $90^\circ$, and construct a control polygon for each using formula (3.9). Then all sub-arcs can be “glued” together in a NURBS representation by identifying the endpoints of adjacent control polygons and modifying the weights and the knot vectors. More precisely, suppose two control polygons are given: $\{P_0, P_1, P_2\}$, and $\{Q_0, Q_1, Q_2\}$, such that $P_2 \equiv Q_0$. Since the two sub-arcs are of equal length, according to formula (3.9) their weight vectors will be identical: $\{1, w_1, 1\}$. The resulting NURBS representation of the concatenated arc will then be:

control polygon: $\{P_0, P_1, P_2, Q_1, Q_2\}$
weight vector: $\{1, w_1, 1, w_1, 1\}$
knot vector: $\{0, 0, 0, \frac{1}{2}, \frac{1}{2}, 1, 1, 1\}$

Concatenating more than two sub-arcs follows a similar pattern with respect to the control polygons and the weight vectors. The knot vector $U$, depending on the number of sub-arcs, or equivalently, the length $\Delta \theta$ of the original arc, is:

$0 \leq \Delta \theta \leq 90^\circ : U = \{0, 0, 0, 1, 1, 1\}$
$90^\circ < \Delta \theta \leq 180^\circ : U = \{0, 0, 0, \frac{1}{2}, \frac{1}{2}, 1, 1, 1\}$
$180^\circ < \Delta \theta \leq 270^\circ : U = \{0, 0, 0, \frac{1}{3}, \frac{1}{3}, \frac{2}{3}, \frac{2}{3}, 1, 1, 1\}$
$270^\circ < \Delta \theta \leq 360^\circ : U = \{0, 0, 0, \frac{1}{4}, \frac{1}{4}, \frac{1}{2}, \frac{1}{2}, \frac{3}{4}, \frac{3}{4}, 1, 1, 1\}$

**NURBS Line Segment Parametrization**

Finally, we mention the NURBS parametrization of straight line segments, which are used as generating curves for annular and conical-type polygons. If the line segment is defined by its end points $P_0$, and $P_1$, then it can be represented as a degree-1 NURBS curve in the following manner:

control polygon: $\{P_0, P_1\}$
weight vector: $\{1, 1\}$
knot vector: $\{0, 0, 1, 1\}$

Notice that this is just a fancy way of writing the familiar line segment affine parametrization: $(1 - u)P_0 + uP_1$, $0 \leq u \leq 1$

**3.1.3 OpenGL’s NURBS Interface**

Using OpenGL’s NURBS interface is relatively straightforward. To draw a NURBS surface patch the following steps need to be performed:

1. Call `gluNewNurbsRenderer()` to generate a pointer to a NURBS object, which is referred to when the NURBS surface is later created.
2. If necessary, call `gluNurbsProperty()` to choose rendering values, sampling method and other properties of a previously created NURBS object.

3. Call `gluNurbsCallback()` in case it is necessary to obtain notification about errors encountered during NURBS rendering.

4. Start the definition of a NURBS surface by calling `gluBeginSurface()`.

5. To generate and render the surface patch, call `gluNurbsSurface()` with arguments: the control points array, the knot vector, and the order of the B-spline basis functions.

6. Call `gluEndSurface()` to complete the definition of the NURBS surface patch.

For example, suppose that control mesh points have been computed and their coordinates stored in a one-dimensional array `ctrlpoints[]` of size `u_ctlpts×v_ctlpts`, where `u_ctlpts`, and `v_ctlpts` denote the number of control points in the `u`- and the `v`-direction respectively. Coordinate values can be packed in either row-first, or column-first order. Notice that these are in fact the “weighted” control points, i.e. a point \( P(x, y, z) \) and its associated weight \( w \) are “upgraded” to a point \( P^w \) in projective space with homogeneous coordinates: \( (wx, wy, wz, w) \).

Suppose also that the knot vectors in the `u`- and the `v`- direction are `Uknot[]` and `Vknot[]`, each of size `u_knotsize`, and `v_knotsize` respectively. We can define and render the corresponding NURBS surface patch with the following command sequence:

```c
type(GLUnurbsObj), pointer :: patch

patch = gluNewNurbsRenderer()
call gluNurbsProperty(patch, GLU_U_STEP, 10_GLfloat)
call gluNurbsProperty(patch, GLU_V_STEP, 10_GLfloat)
call gluBeginSurface(patch)
call gluNurbsSurface(patch, u_knotsize , Uknot(1:), v_knotsize, Vknot(1:), 4.0*v_ctlpts, 4.0_GLfloat, ctrlpoints(1:), U_ORDER, V_ORDER,&
GL_MAP2_VERTEX_4)
call gluEndSurface(patch)
```

The two calls to `gluNurbsProperty` set the number of sample points per unit length taken in the `u`- and `v`-direction. This parameter controls how smooth the patch will look like. The parameters `4.0*v_ctlpts` and `4.0_GLfloat` indicate the offset (as a number of single-precision floating point values) between successive control points in the parametric `u`- and `v`- direction respectively. These particular values mean that each control point is a list of 4 elements, and that there are `4*v_ctlpts` control points along the `u`-direction. Note that the offset parameters must be specified of type `float`, even though they are associated with integer values; this is just one of the idiosyncracies of OpenGL’s NURBS interface. The last parameter, `GL_MAP2_VERTEX_4`, refers to type of map in terms of the dimension of the target space. Implicitly, this determines whether the surface patch is polynomial or rational. Maps into 4-dimensional affine space are interpreted as maps into 3-dimensional projective space via the homogeneous coordinates representation, and hence determine a rational patch.
3.2 Rotations

Rotations are usually among the basic object transformations supported by interactive visualization software. While the commands that rotate an object are very straightforward, the process of converting mouse input into properly accumulating 3D-rotations can be surprisingly complicated.

3.2.1 Quaternion vs. Axis-defined Rotations

One way to represent a rotation is via decomposition into rotations around the coordinate axes. A classical approach is to use all axes, in which case the three angles defining a given rotation are the well-known Euler angles. Another approach - in fact the one implemented originally in the visualization package uses just two axes - say X and Y - and accumulates the angles of rotation around each. Both approaches have their drawbacks: Euler angles are more intuitive but there is no straightforward way to associate them to mouse moves; using them may also cause a phenomenon known as gimbal lock - a special position in which the object loses a degree of freedom and remains “locked” in some isolated region of the configuration space. The other approach has the problem of the interface being not so intuitive, i.e. sometimes mouse moves will rotate the object in an unexpected direction; also, since there are only two axes involved in the representation, manipulating the object could be cumbersome and counter-intuitive in certain situations.

A third approach, based on quaternion representation of rotations, eliminates the aforementioned problems, and provides a mathematically and numerically robust method for accumulating incremental rotations. The theory behind quaternions is beyond the scope of this report; however, a few basic facts can be stated without going into much detail:

1. Quaternions form an algebra over the reals, denoted by $\mathbb{H}$, with basis $1, i, j, k$, i.e. every quaternion $q$ can be (uniquely) written as:

   $$ q = r + xi + yj + zk $$

   The basis elements $i, j, k$ are all square roots of unity, and satisfy a number of relations as a result of which the quaternion algebra is associative but not commutative, i.e. for any $q_1, q_2, q_3 \in \mathbb{H}$:

   $$ q_1 * (q_2 * q_3) = (q_1 * q_2) * q_3 \quad \text{and} \quad q_1 * q_2 \neq q_2 * q_1 $$

2. Quaternions possess a norm, i.e. a “length function”; the length of $q = r + xi + yj + zk$ is:

   $$ |q| = \sqrt{r^2 + x^2 + y^2 + z^2} $$

   If $|q| = 1$ we call $q$ a unit quaternion.

3. A quaternion $q = r + xi + yj + zk$ corresponds to a pair $(s, v)$, where $s \in \mathbb{R}$ and $v \in \mathbb{R}^3$, so that $s = r$ and $v = (x, y, z)$. With respect to this representation, the multiplication formula for two quaternions $(s_1, v_1), (s_2, v_2)$ can be written as:

   $$ q_1 * q_2 = (s_1s_2 - v_1 \cdot v_2, s_1v_2 + s_2v_1 + v_1 \times v_2) $$

32
(in the abstract algebraic jargon, this says that \( \mathbb{H} \) is isomorphic to a semidirect product of \( \mathbb{R} \), and \( \mathbb{R}^3 \)).

The last property makes possible the use of quaternions for representing rotations. Namely, a rotation of angle \( \phi \) around a unit vector \( \mathbf{u} \) corresponds to the unit quaternion \((\cos(\phi/2), \mathbf{u} \sin(\phi/2))\). Concatenating rotations amounts to multiplying their associated quaternions. Thus for an algorithm that implements rotations it suffices to provide the following:

1. A procedure that converts a mouse move into an axis-angle pair representing a rotation. This is tantamount to constructing a trackball simulation model (see below).
2. A procedure that converts an axis-angle pair into the corresponding quaternion.
3. A procedure for multiplying two given quaternions.
4. A procedure that constructs the rotation matrix corresponding to a given quaternion. Eventually, when the graphic objects need to be updated, this procedure would provide the rotational part of the modelview transformation.
5. A data structure that keeps track of the current quaternion; updated each time a mouse move generates a rotation.
6. To ensure numerical stability, a procedure that normalizes a given quaternion. Since conversion between axis-angle and quaternion representations inevitably involves inverse trigonometric functions, after a number of cycles quaternions start to drift away from the unit length. A global counter can keep track of the conversion procedure’s activation record, and reset the length of the current quaternion when a certain predefined limit is reached.

3.2.2 Trackball Simulation

The idea behind a virtual trackball is the following. Eventually, we would like to construct some (one-to-one) correspondence between the points on the mouse pad and trackball points, so that a mouse move would correspond to (the unique) rotation of the sphere that carries the initial point to the end point. Intuitively, this can be accomplished if we “wrap” the mouse pad around the trackball without overlapping, and then figure out a way to translate mouse moves into sphere rotations.

More precisely, a mouse position can be associated with a rank-2 vector, scaled with respect to the window size so that its length is proportional to the distance traveled by the mouse pointer within the window. Suppose the scaling is such that this length is always less than some \( \varepsilon > 0 \), so in effect we get a map from the set of mouse positions onto the set \( D_\varepsilon = \{ v \in \mathbb{R}^2 | \|v\| < \varepsilon \} \).

Fix a point \( p \) on a sphere \( S \) of radius \( r \) and look at the tangent space \( T_p \) at \( p \). Of course, \( T_p \) is isomorphic (as a vector space) to \( \mathbb{R}^2 \), so we can think of \( D_\varepsilon \) as a subset of \( T_p \), centered at the origin. Now we need to construct a map from \( T_p \) to \( S \), which is one-to-one on \( D_\varepsilon \), and is also “geometrically nice” in the sense that it doesn’t distort distances (well, at least by not more than a constant factor). Fortunately, differential geometry has a recipe for this, called “the exponential map”, which actually works in much more general contexts but here can be
described as follows. Let $v \neq 0$ be a vector in the tangent space $T_p$ at a point $p$ of a surface.

There exists (locally) a unique geodesic line $\gamma_v$ on the surface, whose tangent vector at $p$ is equal to $v$, i.e.

\[
(*) \quad \gamma_v(0) = p, \quad \frac{d}{dt} \bigg|_{t=0} \gamma_v(t) = v
\]

The exponential map at $p$ is the function $\exp_p: T_p \to S$, such that $\exp_p(v) = \gamma_v(1)$. When $v = 0$, we set $\exp_p(0) = p$ for all $t \in \mathbb{R}$. We call the point $p$ the base point of $\exp_p$. In the case of the sphere, the geodesic $\gamma_v$ is defined globally and is nothing but the great circle through $p$ that is tangent to $v$ and parametrized so that \((*)\) holds.

For a compact surface, such as the sphere, the exponential map can never be one-to-one globally, but theory guarantees the existence of a neighborhood of $p$ (called normal), for which the map will be one-to-one (and hence a diffeomorphism). For a sphere of radius $r$, such a neighborhood can be easily computed: if $u$ is a unit vector in $T_p$, and $p$ is the unit vector corresponding to the point $p$, both viewed as vectors in $\mathbb{R}^3$, the map $\exp_p$ wraps the radial line through $u$ around the great circle through $p$ to which $u$ is tangent, i.e.

\[
\exp_p(tu) = r[\cos(t/r)p + \sin(t/r)u]
\]

Obviously, the exponential map is one-to-one for $0 \leq t < \pi r$, and hence on the set $D_{\pi r}$. As a special case, we can take $p$ to be on the positive $z$-axis; then $u$ will be a unit vector in the $xy$-plane, and formula 3.10 can be written as follows:

\[
(3.11) \quad \exp_p(tu) = \begin{pmatrix}
 r\sin(t/r)u_x \\
 r\sin(t/r)u_y \\
 r\cos(t/r)
\end{pmatrix}
\]

where $u_x$ and $u_y$ are respectively the $x$- and the $y$-coordinates of $u$.

Suppose now that in a graphics window of width $W$ and height $H$, a local 2D-coordinate system is set up so that the origin is at the window’s center point. Then the set of mouse positions is the rectangle: $[-W/2, W/2] \times [-H/2, H/2]$, which can be inscribed in a circle of radius $\epsilon = \frac{1}{2}\sqrt{W^2 + H^2}$ and hence for $r > \epsilon/\pi$, the exponential map onto a sphere of radius $r$, tangent to the window plane at the point $(0,0,r)$, will be one-to-one. Hence, if we fix an $r$ that satisfies the above inequality, we can compute the projection of a point with coordinates $(X,Y)$ under the exponential map using formula 3.11 by setting:

\[
t = \sqrt{X^2 + Y^2} \quad u_x = \frac{X}{\sqrt{X^2 + Y^2}} \quad u_y = \frac{Y}{\sqrt{X^2 + Y^2}}
\]

Now, a mouse move, i.e. a translation from point $p_1 = (X_1, Y_1)$ to point $p_2 = (X_2, Y_2)$, being an isometry of the tangent plane, turns into an isometry of the sphere, namely the rotation that carries $m_1 = \exp(p_1)$ to $m_2 = \exp(p_2)$. Computing this rotation, given the points $m_1$ and $m_2$, is a routine exercise in vector calculus involving cross products and will not be discussed here.

The only problem with formula 3.11 is that it involves trigonometric functions and square roots, so its computer implementation could be costly and susceptible to numerical imprecision. Fortunately, in this case we can replace the great circle trigonometric parametrization with one
that uses only polynomials, namely the *rational quadratic parametrization* described in section (3.1.1), for a circle of radius $r$, spanned by the point $p = (0, 0, r)^T$ and a unit vector $u$, looks like:

$$C_u(t) = \frac{(1-t)^2w_0P_0 + 2t(1-t)w_1P_1 + t^2w_2P_2}{(1-t)^2w_0 + 2t(1-t)w_1 + t^2w_2}, \quad t \in [0, 1]$$

where:

$$P_0 = \begin{pmatrix} 0 \\ 0 \\ r \end{pmatrix}, \quad P_2 = ru, \quad P_1 = P_0 + P_2$$

$$w_0 = w_2 = 1, \quad w_1 = \frac{\sqrt{2}}{2}$$

Notice that since in our case $u$ belongs to the $xy$-plane, this arc spans only a quarter of a circle, so in effect the image of the exponential map will be contained just in the upper hemisphere of the virtual trackball. This choice of parametrization is justified by the observation that the farther from the base point mouse positions get mapped to, the less intuitive rotations look like. The equator seems to be just the right place to stop.

Eventually, the exponential map based at $p$, written in terms of this particular parametrization is:

$$\exp_p(m) = C_u(t)$$

where $m = (X, Y) \in [-W/2, W/2] \times [-H/2, H/2]$ is the mouse position in the “centered” window coordinates, $u$ is a the normal vector along $m$, and

$$t = \begin{cases} |X|/W & \text{if } |X| > \frac{W}{H}|Y| \\ |Y|/H & \text{otherwise} \end{cases} \quad \Rightarrow t = \max\left(\frac{|X|}{W}, \frac{|Y|}{H}\right)$$

The expression for $t$ simply measures how far from the origin the mouse position is relative to the window boundary, along the radial line through the origin. Consequently, boundary points, which for each radial line correspond to $t = 1$, map to the trackball’s equator. Notice that with this parametrization, the radius $r$ has no effect on the injectivity of the exponential map since the image of the map is apriori restricted within the upper hemisphere by the particular choice of the control points $\{P_i\}$. A reasonable choice for the value of $r$ is, for example, half the diagonal length of the window since in that case distances on the screen and on the trackball are of the same order of magnitude, so rotation feels very natural.

### 3.2.3 Trackball Modification

Our trackball model is based on the methods implemented in the *trackball library* - a set of routines distributed with the GLUT demo source code. We have made, however, two significant changes in the original algorithm:

- We use exponential mapping to project mouse positions to the virtual trackball. In contrast, the method used in the original version of the trackball library projects onto a surface that is partially a sphere and partially a hyperbolic sheet. In particular the projection is along the $z$-axis, so the $x$- and the $y$-coordinates of a mouse position remain
unchanged. To compensate for the non-uniform scaling along radial directions, mouse positions within a certain distance to the origin are mapped onto a sphere, while mouse positions beyond this distance are mapped onto a hyperbolic sheet. This is less geometrically and computationally robust than the exponential mapping method, since the rotations thus generated do not necessarily belong to the isometries of the same sphere. Also, the parametrizations used are non-uniform, so mouse moves of the same length may produce rotations of different angles depending on the relative position of the mouse pointer within the window.

- We have made the trackball projection take into account the current viewpoint. The original implementation assumes that the camera is always positioned on the positive z-axis, looking towards the origin with the y-axis pointing upward. This is usually the default viewing situation but often in OpenGL programming the camera needs to be placed elsewhere and oriented in a non-trivial way. We have developed a method to adjust the exponential projection in order accommodate such cases; a description of this method is given below.

Clearly, the exponential projection formula looks particularly simple when the base point is on the positive z-axis and the remaining premises of formula 3.11 are satisfied. So if we transform the current viewing frame into “standard position”, we can apply formula 3.11 and obtain the projections of the modified mouse coordinates. This initial transformation is just a rotation of \( \mathbb{R}^3 \), and can be expressed in terms of an orthogonal \( 3 \times 3 \) matrix \( M \). Suppose \( \mathbf{u} \) is a mouse position in the original viewing frame, and \( p \) the base point of the exponential map. The transformed position and base point in the “standard frame” are then \( \mathbf{u}' = M \mathbf{u} \) and \( p' = (0, 0, r)^T \) respectively. Now the image of \( \mathbf{u} \) under the exponential map \( \exp_p \) can be written as:

\[
\exp_p(\mathbf{u}) = M^{-1} \circ \exp_{p'} \circ M \mathbf{u} = M^{-1} \circ \exp_{p'}(\mathbf{u}')
\]

Thus we only need to compute \( M^{-1} \) once per viewframe change and then use it each time to obtain the projected mouse coordinates. To set a new viewframe, OpenGL has a special utility function called \texttt{gluLookAt} and it is not hard to observe that \( M^{-1} \) can be read off the modelview matrix generated by \texttt{gluLookAt}. Indeed, the latter is a \( 4 \times 4 \) matrix which rotates the standard frame to the specified one and translates the origin (of the global coordinate system) to the desired eye position. The rotational part of the modelview matrix is nothing but \( M^{-1} \) and can be extracted by reading the upper left \( 3 \times 3 \) block. Thus the only overhead in computing \( M^{-1} \) is the single operation that copies a matrix from the modelview matrix stack. The following code segment illustrates how this can be done in OpenGL:

```c
!!# eye(3) defines the viewpoint, center(3) the point at which the
!!# camera is aimed; together they define the viewing direction;
!!# up(3) is the vector that defines the vertical direction for the
!!# camera; trans(16) is a 1-dimensional array into which the matrix
!!# generated by gluLookAt is copied in unpacked form (along rows).

call glMatrixMode(GL_MODELVIEW)
call glLoadIdentity
call gluLookAt(center(1), center(2), center(3), &
```
eye(1), eye(2), eye(3), &
up(1), up(2), up(3))
call glGetDoublev(GL_MODELVIEW_MATRIX, trans)

!!# We only need the rotational part of the transformation
!!# matrix, i.e. the upper left 3x3 block

\[
\begin{align*}
\text{tm}(1,1:3) &= \text{trans}(1:3) \\
\text{tm}(2,1:3) &= \text{trans}(5:7) \\
\text{tm}(3,1:3) &= \text{trans}(9:11) \\
\end{align*}
\]

!!# since trans is generated eventually by a C routine it contains the 
!!# transformation matrix in row-major form. To use the matrix in
!!# Fortran we need to convert to column-major form, i.e. transpose.
\[
\text{tm} = \text{Transpose}(\text{tm})
\]

### 3.2.4 Rotation Around the Domain’s Center Point

In our earliest implementation each axis of rotation was assumed to pass through the origin of the modeling coordinate system, i.e. the origin was the fixed point of every rotation. As a result, a data domain that happened to be away from the origin, was rather inconvenient to manipulate. To illustrate this point, imagine that a unit-size cube is centered at \((5, 0, 0)\) and that the camera is located at \((5, 0, 10)\) pointing at \((5, 0, 0)\), i.e. the cube appears at the center of the viewport window. Unless the field of view is unreasonably large, there would always be rotations around the \(y\)-axis that move the cube outside of the viewing volume. Thus, in order to see the opposite face of the cube, one would have to either rotate only around the \(x\)-axis (which in the current setting is practically impossible, unless one has the hand precision of a brain surgeon!), or apply a combination of rotations, translations and/or zoomouts. Clearly, the best choice for a rotational fixed point in a single-object scene is, in general, the object’s center, which in our situation can be computed in a straightforward way since the data domain is either a rectangular box, or a spherical wedge. In particular, the domain is defined as:

\[
[\xi_1^{\text{min}}, \xi_1^{\text{max}}] \times [\xi_2^{\text{min}}, \xi_2^{\text{max}}] \times [\xi_3^{\text{min}}, \xi_3^{\text{max}}]
\]

where \((\xi_1, \xi_2, \xi_3)\) are domain coordinates, which can be either Cartesian or spherical. Hence the center, given in domain coordinates, is the point:

\[
P = \left( \frac{\xi_1^{\text{min}} + \xi_1^{\text{max}}}{2}, \frac{\xi_2^{\text{min}} + \xi_2^{\text{max}}}{2}, \frac{\xi_3^{\text{min}} + \xi_3^{\text{max}}}{2} \right)
\]

In the Cartesian case, the \(\{\xi_1, \xi_2, \xi_3\}\) coordinates are identical with global coordinates \(\{x, y, z\}\), so the center is precisely \(P\). In the spherical case, \((x, y, z) = \sigma(\xi_1, \xi_2, \xi_3) = \sigma(r, \phi, \theta)\), where \(\sigma\) is the spherical coordinate transformation map, and thus the center is \(\sigma(P)\). The following OpenGL segment illustrates how rotation around a point with coordinates \((\text{mp}(1), \text{mp}(2), \text{mp}(3))\) can be performed:

\[
\begin{align*}
call glTranslatef(\text{mp}(1), \text{mp}(2), \text{mp}(3)) \\
call glMultMatrixd(m) \\
call glTranslatef(-\text{mp}(1), -\text{mp}(2), -\text{mp}(3))
\end{align*}
\]
Here \( \mathbf{m} \) is the rotation matrix obtained from the current quaternion by the build\_matrix trackball library routine.

### 3.3 Viewing Transformations

Choosing an appropriate camera position and orientation is one of the important aspects of scene generation in graphics programming. By default, OpenGL places the camera at the origin, looking down the negative \( z \)-axis, and oriented so that the \( y \)-axis is pointing upward. To see an object positioned, for example, along the positive \( z \)-axis, i.e. behind the camera, we have to either move the object, or move the camera. The former serves as an example of a \textit{modeling transformation}, whereas the latter is called a \textit{viewing transformation}. The two types of transformations are, in a sense, dual to each other in that each has a counterpart of the other type that produces the same result. This provides the graphics programmer with greater conceptual flexibility and allows for various approaches to scene modeling.

Given the specific nature of our visualization problem, we have taken the following approach to the modelview setup. A number of “default views” have been precomputed, each with a different camera position and orientation. In particular, three of them have the camera placed along the coordinate axes, and eight more - along the diagonals of a cube centered at the origin. In all views, except one, the camera is pointed towards the origin and the up-vector is aligned with the positive \( z \)-axis. The exception is, of course, when the viewpoint is on the \( z \)-axis itself, in which case the up-vector is aligned with the \( y \)-axis. Each view can be selected via the pop-up menus and is initialized via a call to the view setup function, which carries out the following tasks:

1. Set up the respective right-oriented local orthonormal frame at the camera position. That frame is later used to obtain the up-vector for the call to OpenGL’s \texttt{gluLookAt} as well as to provide a vector basis for user-defined translations.

2. Compute the corresponding trackball matrix \( \mathbf{tm} \) and initialize the trackball simulation interface.

3. Set the translation parameters so that the center of the data domain gets mapped initially to the origin by the modeling transformation. This means that, in effect, the domain will appear centered inside the viewport.

4. Compute the appropriate distance from the camera position to the origin along the line of sight. This distance must be such that after any rotation, domain fits the size of the viewport along its largest diameter.

5. Call \texttt{gluLookAt} with the arguments obtained in the previous steps to initialize the modelview matrix and push it onto the matrix stack (see 3.2.3).

Once a default view has been set up, only modeling transformations, namely user-defined rotations and translations, are applied to the scene.

The following sections elaborate on two of the above listed items (1 and 4):

\footnote{For the “diagonal cases” this means that the angle between the up-vector and the \( z \)-axis is less than 90°}
3.3.1 The Local View Frame

This frame consists of three orthonormal vectors \{\mathbf{e}, \mathbf{u}, \mathbf{r}\}, where \mathbf{e} is along the line of sight, \mathbf{u} is mapped by the viewing transformation (via \texttt{gluLookAt}) to the positive \(y\)-axis so it will point upward in the viewport, and \mathbf{r} complements the first two to a right-oriented frame. The latter implies that \mathbf{r} will be mapped via \texttt{gluLookAt} to the positive \(x\)-axis, i.e. it will point to the right in the viewport.

The local view frame is used to determine the proper direction for user-defined translations and zoom-ins/outs. In particular, a mouse move with (normalized) window coordinates \((x, y)\) generates a translation defined in global coordinates by the vector \(x\mathbf{r} + y\mathbf{u}\). This means that regardless of the camera position, translations will always be perpendicular to the line of sight and “in sync” with the mouse motion. Similarly, zooming in or out is achieved via translations defined by the vector \(k\mathbf{e}\), where \(k\) is a scalar that is positive when zooming out, and negative otherwise.

3.3.2 Domain Fits the Viewport

To compute the how far the camera should be from the origin we need to take into account the size of the domain along its maximal diagonal, as well as the camera’s field of view and aspect ratio. The maximal diagonal is defined as the line segment between a pair of domain points that are at maximum distance from each other. For instance, in a rectangular domain this will be any of the four body diagonals. In the spherical case, since the domain is the image of the rectangular box: \([r_{\min}, r_{\max}] \times [\phi_{\min}, \phi_{\max}] \times [\theta_{\min}, \theta_{\max}]\) under the spherical coordinate change map \(\sigma\), a maximal diagonal will be, for example, the line segment between the points \(\sigma(r_{\min}, \phi_{\min}, \theta_{\min})\) and \(\sigma(r_{\max}, \phi_{\max}, \theta_{\max})\).

In either case, there exists a sphere \(S\), with diameter equal to a maximal diagonal, that contains the entire domain and its images under rotation (this is certainly false in general but in our situation holds true). Consider now the viewing volume created by the perspective projection setup. It is a frustum with vertex at the camera position, and whose smaller base is mapped onto the viewport. The angle between the lower and the upper clipping plane is the camera’s field of view \(\alpha\). Suppose the aspect ratio \(r\) (=width/height) is greater than 1. This means that the viewing volume is smaller in the vertical direction. The domain fitting problem is equivalent to the following: at what distance from the frustum’s vertex is the sphere \(S\) tangent to the upper and the lower clipping planes?

We can easily translate this into a planar problem if we consider a vertical cutting plane along the line of sight (fig. 3.3). Clearly, the desired distance is equal to \(d/(2\cos(\alpha/2))\), where \(d\) is the diameter of \(S\). Similarly, when \(r \leq 1\) this distance is equal to \(d/(2\cos(\beta/2))\), where \(\beta\) is the horizontal vertex angle for the viewing frustum. It is not hard to see that the relation between \(\beta\) and the field of view angle \(\alpha\) is: \(\tan(\beta) = r \tan(\alpha)\).

3.4 Polygon Offset

An earlier implementation supported an option that toggled the display of the cell and block boundaries. To highlight a polygon boundary, the polygon was rendered in two different rasterization modes - once in filled, and once in wireframe mode. Because lines and filled polygons
(a) Viewing frustum and the domain enveloping sphere $S$

(b) Planar cross-section through the line of sight $VO$

Figure 3.3: Computing the camera position for domain fitting
are not rasterized in exactly the same way, depth-buffer values of otherwise coincident objects may turn out to be different. The highlighting lines may fade in and out of a polygon, creating an unpleasant visual artifact that is sometimes referred to as “stitching”. This problem was handled in that implementation via disabling the depth buffer comparison mode when displaying cell or block boundaries. Needless to say, such a method has very limited application, since rendering more than one non-coplanar object without hidden surface removal is practically useless.

OpenGL offers a nice solution to this problem via a polygon offset mechanism, which adds an appropriate offset to polygon or line primitives in order to force coincident z-values apart to cleanly separate a polygon edge from its highlighting line. Polygon offset is turned on by passing the appropriate parameter to `glEnable()` - either `GL_POLYGON_OFFSET_FILL`, or `GL_POLYGON_OFFSET_LINE` - depending on which primitives we would like the offset to be applied to. There are two ways to get rid of the “stitching” artifacts: we can either add positive offset to the filled polygons to push them away from us, or add negative offset to the wireframe to pull it towards us. The following command specifies the amount of offset to be applied to rasterized objects:

```c
void glPolygonOffset(GLfloat factor, GLfloat units)
```

The offset value is a linear function whose coefficients are given by `factor` and `units`, namely:

\[
    offset = m \cdot factor + r \cdot units
\]

where \( m \) is the maximum depth slope of the polygon and \( r \) is a machine specific constant that has to do with screen resolution. The depth slope is the change in \( z \)-\( \text{(depth)} \) values divided by the change in either \( x \)- or \( y \)-coordinates as the polygon is traversed along its edges. Thus polygons that are close to being parallel to the near and far clipping planes have depth slope of 0, while those that are close to being perpendicular have increasingly larger depth slopes. Unfortunately, since various types of polygons may happen to be displayed at once and since polygons can be rotated at any time by the user, it is very difficult to find a single set of values for `factor` and `units` that fit all situations. Thus, the user is given the option for fine-tuning the offset values via keyboard input. In particular, a designated pair of keys increments/decrements the current value of `factor` by a certain amount; the value of `units` is treated similarly.

### 3.5 Selection Mode

**Introduction** An important feature of any scientific visualization software is the ability to generate data feedback from an object’s screen representation. In the case of VizFlow we have a spatial domain partitioned into a grid of cells, with each of which there is an associated array of parameters. At any given time, cells are displayed as filled polygons whose color correlates with the value of the currently selected parameter. While the coloring scheme conveys some information about the relative distribution of parameter values throughout the entire domain, very often the user needs to look at the precise value (or values) associated with a given cell. One approach, taken in an earlier implementation, is to display, at the user’s request, the value of each cell by simply placing it next to the cell. There are several drawbacks of this method, e.g. characters overlap over densely packed domain regions; some values can be incorrectly
associated with an adjacent cell; characters are usually non-scalable, so object transformations
may produce rather unexpected results etc.

A much more robust approach is to use a feature of OpenGL that allows points on the screen
to be mapped back to the points in the modeling space from which they were projected. Thus
when the user clicks on a point inside the graphics window, it is possible to trace back the cell
to which the original point belongs, and display - in a text window - the values of all parameters
associated with that cell. Here is how this is implemented in the VizFlow code.

**Technical Details**  When the cursor coordinates + depth value at the current cursor position
are used to determine the global coordinates of the corresponding 3D-point, we say that the
program is in *selection mode*. No transformations are allowed while in this mode to avoid
asynchronization between the data obtained from the frame buffer and the current display.
Suppose that the mouse has been clicked at a given point. First, the viewport information, the
*modelview*, and the *projection* matrices are obtained. Then the depth value is read back from
the depth buffer. Finally the global (domain) coordinates are calculated with the *gluUnProject*
command. The following code segment illustrates these ideas:

```fortran
!# note that viewport(4) is height of window in pixels
GLy = viewport(4) - y - 1_GLint
realy = Real(GLy, kind=GLdouble)
call glReadPixels(x, GLy, 1_GLsizei, &
  1_GLsizei, GL_DEPTH_COMPONENT, GL_FLOAT, depth)
realz = Real(depth(1), kind=GLdouble)
print *, "Coordinates at cursor are (" , x, realy,")"
success = gluUnProject (realx, realy, realz, &
  mvmatrix, projmatrix, viewport, &
  wx, wy, wz);
if (success .ne. 0) then
  if (datatype==2) then
    !# in case the mode is spherical (datatype=2), world cartesian
    !# coordinates need to be converted to back to spherical
    rho = DSQRT(wx*wx + wy*wy + wz*wz)
    theta = DACOSD(wz/rho)
    if (theta < epsilon .OR. theta > 180.0 - epsilon) then
      print *, "Theta is too close to 0.0 or 180.0"
      return
    endif
    temp = rho*DSIND(theta)
  endif
  endif
endif
```

!!# Will assume that -90.0 <= phi <= 90.0; otherwise there is
ambiguity between phi and 360.0 + phi.
phi = DASIND(wy/temp)
wx = rho; wy = theta; wz = phi
end if
print *, "domain coords at z=",depth," are ", &
("",wx, wy, wz, ")"
call locate_cell(wx, wy, wz)
else
print *, "Can’t UnProject: non-invertible matrix"
end if

Notes:

• The cursor position coordinates \((x, y)\) are of type \texttt{GLint}, so they have to be converted to \texttt{GLdouble}.

• The \(y\)-cursor position is measured from the top of the window (in X windows), while the OpenGL \(y\)-coordinate is measured from the bottom, so we need to convert \(y\) to \(GLy = \text{viewport}(4) - y - 1.\) \texttt{GLint}. (\texttt{viewport}(4) is the window’s height)

• The depth component is obtained by reading the element at position \((x, GLy)\) of the Depth Buffer. Note that the data is read into a pixel array, so the variable \texttt{depth} has to be of array type - in this case a single-element, one-dimensional array. Also the constants that determine the array size - \((1 \times 1)\) - must be of type \texttt{GLsizei}.

• \texttt{gluUnProject} is actually a function, so in Fortran it cannot be called as a subroutine. The result returned by \texttt{gluUnProject} is of type \texttt{GLint} and indicates the success of its execution.

• The function \texttt{locate_cell} searches through the data domain to locate the cell that contain the point with given coordinates.

### 3.6 Menus

An earlier implementation provided only keyboard input for its interactive user interface. In VizFlow, this interface has been extended with pop-up menus - a common feature in almost any modern graphics application. We have used GLUT’s cascading menu API, which creates rather plain looking but simple for setup and maintenance menus. GLUT’s menu management mechanism is straightforward: menus are created, changed, and “attached” to a mouse button within a window. When a menu entry is selected, the callback function registered with that entry will be called. The following example sets up a pop-up menu with two options that can be used to toggle the grid on/off in VizFlow:

```fortran
grid = glutCreateMenu(grid_menu)
call glutAddMenuEntry("On", 1)
call glutAddMenuEntry("Off", 2)
call glutAttachMenu(GLUT_RIGHT_BUTTON)
```
Note that `glutCreateMenu` returns an integer value, which can be used as an entry index in another menu for creating nested entries. The callback function `grid_menu` may look like this:

```fortran
subroutine grid_menu(choice)
   integer :: choice
   select case(choice)
   case (1)
      SelectMode = .TRUE.
   case (2)
      SelectMode = .FALSE.
   end select
end subroutine grid_menu
```

We have duplicated most of the earlier implementation’s keyboard functionality via menus. In addition to that, menu entries have been created for all new features implemented. Unfortunately, GLUT does not provide any other GUI widgets, such as dialog boxes, buttons etc., that are characteristic of sophisticated API’s like Motif, Athena, and Tcl/Tk among others. The future development of VizFlow will involve replacing GLUT with some of these advanced API’s in order to create a more professional “look and feel” for the graphic user interfaces.

### 3.7 Performance Issues

In an earlier implementation, data structures were recomputed and geometric primitives were rendered at each display cycle even if they had not changed from the previous cycle. This meant that a large number of redundant commands were being executed, thus reducing the program’s performance. As the number of cells in a data domain increases, this overhead starts to become noticeable, especially when more than one cross-sectional slice is being displayed. In spherical mode the situation is exacerbated even further by the additional NURBS data structures and the graphics intensity of the OpenGL’s NURBS interface. To compensate for the increased toll on system resources, we have made two major performance optimization steps: organizing geometric primitives into display lists, and implementing a more efficient display update mechanism.

#### 3.7.1 Display Lists

A display list is a group of OpenGL commands that have been stored for later execution. Since the commands are compiled, parameters evaluated and their values copied into the display list, execution can be significantly accelerated, depending on the graphics hardware and a variety of other factors.

Each display list is identified by an integer index. In order to avoid duplication, an unused index can be obtained via the `glGenLists()` command. A display list is marked by the commands `glNewList()` and `glEndList()`. The following code segment provides a simple example of display list initialization:

```fortran
polygonlist = glGenLists(1)
call glNewList(polygonlist, GL_COMPILE)
```
....

call glEndList()

The second argument of glNewList() indicates that the OpenGL commands will only be compiled but not executed as they are placed in the list. To execute the list at a later time we use the command:

call glCallList(polygonlist)

In VizFlow two display list indices are maintained: one for the list of filled polygons and one for the list of wireframe polygons (used to represent cell and block boundaries). Each time these polygons need to be updated, the same index is reused, effectively overwriting the old display list with the new one. The reason for keeping filled and wireframe polygons in separate lists is to allow switching between grid modes without updating the entire display list structure.

3.7.2 Efficient Update Mechanism

As mentioned before, data structures and geometric primitives need not be recomputed at each display cycle. Two boolean parameters, VarUpdate and GeomUpdate, have been introduced to keep track of the two types of update events. If VarUpdate is TRUE then data structures are recomputed to reflect the latest change in the current plot mode. This would happen, for example, if the user switches from, say, plotting 'X-velocity' to plotting 'Field Density'. The other parameter, GeomUpdate, indicates that the polygon lists representing the data domain need to be updated. This is necessary practically each time when a data structure update has occurred as well. In some situations, however, only geometric primitives are updated, for example when new cross-sectional slices are defined but the plot mode remains the same.

3.8 Axes Display Mode

An optional mode which added to VizFlow allows for the current coordinate frame to be displayed in a separate viewport within the graphics window. By “current coordinate frame” we mean the image of the standard coordinate frame under the current modelview transformation. This provides the user with an important visual cue about the current orientation of the displayed objects.

The Axes Display Mode is implemented as follows. As part of the program’s initialization sequence, line segments representing the x-, y-, and z-coordinate axes are created and placed in a display list. Each line segment is assigned a different color and marked with a letter indicating the axis which it represents. Next, a display list containing the viewport and projection modification sequence is created. The viewport needs to be modified so that the axes appear in a fixed box within the graphics window without being affected by translations. Projection mode is changed from perspective to orthographic in order to avoid angular and linear distortions. The following code segment demonstrates how this is done:

axes = glGenLists(1);
call glNewList(axes, GL_COMPILE);
call glPushAttrib(GL_VIEWPORT_BIT);
call glViewport(0, winpercent*H, (1-winpercent)*H, (1-winpercent)*H);
call glMatrixMode(GL_PROJECTION);
call glPushMatrix();
call glLoadIdentity();
call glOrtho(-10.0, 10.0, -10.0, 10.0, -10.0, 10.0);
call glMatrixMode(GL_MODELVIEW);

call glCallList(coordaxes);

call glMatrixMode(GL_PROJECTION);
call glPopMatrix();
call glPopAttrib();
call glMatrixMode(GL_MODELVIEW);
call glEndList();

Here, it is assumed that the coordaxes display list containing the axes rendering sequence has already been created. The viewport defined with the glViewport command is given in terms of percentages of the graphics window’s width $W$ and height $H$. For example, $\text{winpercent} = 0.8$ indicates that the axes display viewport will be a square of side length $0.2 \cdot H$, occupying the upper left corner of the graphics window. To ensure that the current viewport and projection matrix values will be restored after executing this sequence, they are pushed and subsequently popped out of OpenGL’s attributes and matrix stacks respectively.

During each display cycle, the axes list is called right after loading the rotation matrix $\mathbf{m}$ (computed by the virtual trackball interface; see section 3.2.2), before any translations have been applied. This ensures that the orientation of the current coordinate system will always be “in sync” with the objects in the main window, while its origin stays fixed within the axes viewport.
Chapter 4

User’s Guide

4.1 Getting Started

The best way to start VizFlow is via the `vizflow` init script. This will open a default VizFlow window as well as a new console terminal window for text based interaction with the program. Alternatively, running the `vizflow.bin` executable directly will start the program and the current window shell will be used for text input and output.

At startup VizFlow attempts to read the default data file `data.vf`. If that file is not present in the current directory, the initial display window will be blank. Since data will most likely be kept in separate directories it is reasonable to maintain the `data.vf` file as a symbolic link to the currently (and most frequently) used data set. See Section 4.8 below for information on how to import a new data set within VizFlow.

4.2 Modes of Operation

VizFlow has two major modes of operation: **Display** and **Feedback**. In **Display Mode**, objects representing a data domain can be viewed and manipulated. In **Feedback Mode**, clicking on a point inside the domain reports all data associated with that point, e.g. its position, values of associated variables etc. Switching between modes can be done via the menus or via a specially assigned toggle key.

In addition, several modes for displaying data are available. The default is **Raw Data**, whereby all variables are considered constant within each cell in the domain, and thus their values represent the “true” numerical solution at the center of that cell. In **Raw Data** mode all cells are shaded uniformly. Alternatively, in **Interpolation Mode** data is interpolated across the domain with some prescribed resolution and polynomial degree resulting in a “smoother” but possibly less accurate picture.

In case cell boundary data are available, it is possible to compute and display field lines by specifying a set of initial points in the domain (within a special data file - cf. Section 4.10). This can be done in the **Field Lines Mode**. **Interpolation** and **Field Lines** work independently from one another thus each mode has a separate toggle - both under the Main Menu and at
4.3 Navigation and Object Manipulation

VizFlow’s user interface allows for both mouse and keyboard interaction. The three mouse buttons are associated with the following functions:

**left** In *Display Mode*, translate an object within the graphics window. In Feedback Mode, select a point for data feedback.

**middle** Rotate an object within the graphics window.

**right** Activate the pop-up menu window.

Translations are performed along the direction of the mouse motion, i.e. they are always parallel to the camera view plane. Rotations have a fixed point at the center of the object, and axis that is parallel to the camera view plane and perpendicular to the direction of the mouse motion. This behavior can be interpreted as a simulation of a virtual trackball. Other viewing transformations can be activated via the keyboard or the menus.

**Zooming** Pressing the z/Z key zooms in/out the current view by a constant factor. The axis of zooming is perpendicular to the camera plane and passes through the window center point.

**Preset Views** The current view point can be selected among a set of pre-defined ones. There are eight of those situated along the diagonals of a cube centered at the domain’s origin. Also, there is one default view point on each of the coordinate axes. The set of pre-defined view points can be accessed under Views in the Main Menu.

4.4 Displaying Grids

By default VizFlow displays cells as shaded polygons without boundary. This feature can be modified by the g toggle. Pressing g cycles through the three modes:

no grid → block grid → cell grid

Grids are displayed as black lines superimposed over the polygons representing blocks and cells respectively.

Because of the special way lines and polygons are handled by the graphics interface, it may happen that grid lines become darker, or dimmer, or even completely disappear, depending on the viewing situation. The user has control over this behavior by modifying a pair of parameters, called *polygon offset* parameters. The idea is that polygon interiors are offset from the boundary lines by distance $d$ equal to:

$$d = m.s + r.u$$

where $s$ is the maximum depth slope of the polygon and $r$ is a constant that depends on the hardware implementation. The parameters $m$ and $u$ corresponding to the linear/constant
coefficient respectively in the above equation can be manipulated by the user. It is difficult
to prescribe an algorithm for achieving optimal effect for each viewing situation but a rule of
thumb is to start with \( m = 0 \), and \( u = 1 \), then modify \( u \) until lines are fairly visible, and finally
adjust \( m \) in order to fine-tune the picture. In most cases, however the default values will suffice
and no further adjustments would be necessary. The following keys are used to manipulate the
offset parameters:

\[ M/m \quad \text{Increase/decrease the linear factor in increments of 0.1.} \]
\[ U/u \quad \text{Increase/decrease the constant factor in increments of 0.01.} \]

4.5 Slices

VizFlow displays up to three planar slices of the data domain at a time. The slices are given as
level sets of the coordinate functions. By default three slices intersecting at the domain’s center
are displayed. The Slices option in the Main Menu allows the user to specify different slices by
supplying a value of the corresponding \( x \)-, \( y \)-, or \( z \)-level respectively. To switch a specific slice
on and off use the Toggle menu under Slices.

A particular idiosyncracy at the moment is that when spherical VizFlow data is displayed, slices
are still being indicated as level sets of the \( x \), \( y \), and \( z \) coordinate functions. Regardless of what
their names suggest, however, these functions are actually interpreted by VizFlow as \( r \), \( \theta \), and
\( \phi \) respectively. To eliminate this confusion, the next VizFlow release will feature a Slices Menu
that will be programmed to reflect the current data type, with the coordinate function names
changing accordingly.

4.6 Status Bar

By default the top portion of the main window is occupied by VizFlow’s Status Bar, which is
used for displaying some of the current operational parameters. The left corner of the status bar
displays a unit cube with one vertex at the origin (of the global Cartesian coordinate system),
and sides parallel to the \( x \), \( y \), and \( z \)-axis respectively. This cube rotates in sync with the data
domain and thus indicates the current view point relative to the global coordinate system.
The middle section of the status bar contains information about the name of the current data
file and the name of the currently displayed variable. It also contains a color map indicator,
which shows the progression of colors and their correspondence to the values of the currently
displayed variable. The Status Bar can be switched on and off either via the F2 function key,
or the Modes->Dashboard option under the Main Menu.

4.7 Saving Data

Current Status The user has the option to save the parameters associated with a particular
viewing situation to a file and then recreate that situation at a later time. This is particularly
useful when a number of viewing transformations and various other adjustments have been
applied to the scene, producing an optimal rendering that is worth revisiting in the future.
The default file to which parameters are saved is `current.par` in the current working directory. At start time VizFlow checks for the presence of this file and if it exists, VizFlow uses it to initialize the state of the system parameters. To save the current status, use the `v` key, or the Save->Current Status option under the Main Menu.

**Screen Snapshots** The contents of VizFlow’s graphics window can be dumped to a file for image processing, printing etc. The file is in raw pixel format, i.e. a rectangular array of RGB values and can be manipulated with any standard image processing software. The file’s name starts with the string “screen_save.”, and ends with a suffix “.rgb”. In between is a number that identifies the frame number for the current screen shot. Thus the first snapshot will be named “screen_save_0001.rgb”, the second “screen_save_0002.rgb” and so forth. The frame numbering is relative to the current VizFlow session, thus each time the program is started, frame numbers will be initialized to 0001 and old snapshots with the same names will be overwritten. In the future this behavior will be modified to accomodate a more flexible approach to snapshot file names that will allow the user to specify a base identifier associated with a given series of snapshots.

### 4.8 VizFlow Data

VizFlow assumes data files in a certain format, which we call *VizFlow Internal Format*, or simply *vf Format*. In addition, VizFlow supports (at this time) two external data formats: BATSRUS and AMRFCT. For each of these a Perl script is used to convert a sequence of data files generated by the respective numerical solvers into VizFlow’s internal data format. The Perl script can be run either separately from the command line, or from within VizFlow. When used from the command line, the following syntax applies:

```bash
convert <frame #> <data dir> [AF]
```

- `<data dir>` is the directory containing the corresponding data set.
- `<frame #>` is the frame number of the associated data files
- `AF` is an optional argument indicating that the data set contains files in AMRFCT format.

Alternatively, the script can be activated from the Read Data option under VizFlow’s Main Menu in which case the user is prompted to enter values for the aforementioned command line arguments.

In a parallel environment data associated with each frame is split among several processors each of which outputs its data to a separate file. For each frame the `convert` script cycles automatically through all processor-specific files and assembles the data into a single file in vf format.
4.8.1 Internal Data Format

Following is a description of the VizFlow internal format. Each file starts with a header string, indicating the type of data and the presence of face-centered magnetic field values:

```
#VizFlow Data#[Spherical]#[MagFace]#
```

The “VizFlow Data” string is mandatory, while the other two are optional. The next two lines contain the values of domain-specific variables:

```
ndim nvar nxtot nytot nztot nblocks
```

```
xmin xmax ymin ymax zmin zmax
```

where

- **ndim** is the number of dimensions, either 2 or 3
- **nvar** is the number of variables per cell
- **nxtot, nytot, nztot** are the numbers of cells per block in the \(x\), \(y\) and \(z\) direction respectively.
- **nblocks** is the total number of blocks in the given data file.
- **xmin, xmax** etc. specify the domain’s dimensions

The rest of the data file consists of data clusters each of which corresponds to a block in the grid decomposition of the physical domain. Each data cluster has the following format:

```
nodetype
```

```
center_x center_y center_z
```

```
size_x size_y size_z
```

```
nxtot*nytot*nztot lines of cell-centered data
```

```
[(nxtot+1)*nytot*nztot lines of face-centered x-components]
```

```
nxtot*(nytot+1)*nztot lines of face-centered y-components
```

```
nxtot*nytot*(nztot+1) lines of face-centered z-components
```

where

- **nodetype** indicates the block type with respect to the octree decomposition hierarchy: 1 for a leaf node and 2 for an intermediate node.
- **center_x** etc. are the coordinates of the block’s centroid.
- **size_x** etc. are the side lengths of the block in each dimension.
• each of following $n_{tot}*n_{ytot}*n_{ztot}$ lines of data corresponds to a cell and contains all $n_{var}$ values of the variables computed over that cell

• in case the MagFace string in the header is present, the remaining portion of the block’s data cluster should contain the values of the corresponding face-centered magnetic field components. Each set of values is ordered in a nested-list fashion: with one list for each $x$, $y$, and $z$-component respectively. For instance let $b_x(i, j, k)$ denote the $x$-component of the magnetic field at the $(i, j, k)$-th face in a given block; then the following Fortran code excerpt shows the order of nesting when data are output to the VizFlow file:

```fortran
do k = 1, nztot
  do j = 1, nytot
    do i = 1, nxtot + 1
      print *, b_x(i, j, k)
    enddo
  enddo
enddo
```

4.8.2 AMRFCT Format

AMRFCT data files share the same syntax with vf data files. There are two differences, however: one is that a single AMRFCT file contains data from one of possibly many processors on which the numerical solver had worked to generate the data. The other difference is in the file names: whereas VizFlow doesn’t assume any specific name convention for its internal data files, AMRFCT files are required to have names as follows:

`vf_pXXX_sYYYY.dat`

where $XXX$ is the processor number specified in three-digit format (i.e. along with the leading zeros), and $YYYY$ is the frame number in five-digit notation.

4.9 Movie Capabilities

Besides the usual interactive mode, VizFlow has a special “batch” mode which allows for external programs to call VizFlow with specific processing requests. In particular, included with the program is a Perl script that generates a series of snapshots from a given data set by cycling through the data files and executing VizFlow in batch mode. In each instance, the program reads the current data file, opens a graphics window, takes a snapshot, saves it to an image file and exits. The sequence of image files thus generated can be processed further and converted into a “movie” by means of any standard animation software. The Perl script is activated with the following command line options:

`movie <data dir> <first frame #> <last frame #> <step>`

• `<data dir>` is the directory containing the VizFlow data set.

• `<first frame #>` is the frame number of the first data file
• \(<\text{last frame} \, \#>\) is the frame number of the last date file

• \(<\text{step}>\) is the increment step between successive frames.

To generate a movie using a previously saved system state, simply copy the associated status parameter file current\_par to the current working directory.

4.10 Tracing Field Lines

In order to trace field lines, VizFlow needs two additional data structures: one is the face-centered magnetic field data, and the other, the set of initial points for the trace lines. As mentioned before (cf. Section 4.8), the first data structure can be supplied within the VizFlow data file. The set of trace lines initial points are defined in a separate file called a Trace Init File. The default file name for this file is “traceinit.dat”. In case the latter is not present in the current working directory at startup time, an alternative file should be imported via the Imports menu before switching to Field Lines Mode. Attempting to trace field lines without having supplied both face-center magnetic field data and a set of initial points will produce an error message and no lines will be displayed.

The file where trace lines initial points are defined has the following format:

\[
N_{\text{forward}}
\]
\[
xf[1], yf[1], zf[1]
\]
\[
...
\]
\[
xf[N_{\text{forward}}], yf[N_{\text{forward}}], zf[N_{\text{forward}}]
\]
\[
N_{\text{backward}}
\]
\[
xb[1], yb[1], zb[1]
\]
\[
...
\]
\[
xb[N_{\text{backward}}], yb[N_{\text{backward}}], zb[N_{\text{backward}}]
\]

where

• \(N_{\text{forward}}\) and \(N_{\text{backward}}\) denote the number of initial points that will be used for forward and backward tracing (along the line parameter) respectively.

• \(xf[i], yf[i], zf[i]\) for \(i = 1, \ldots, N_{\text{forward}}\), are the coordinates of the forward initial points

• \(xb[i], yb[i], zb[i]\) for \(i = 1, \ldots, N_{\text{backward}}\), are the coordinates of the forward initial points
4.11 Summary of Command Key Shortcuts

<table>
<thead>
<tr>
<th>Switching Display Variables</th>
<th>Key</th>
<th>Synopsis</th>
<th>Key</th>
<th>Synopsis</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>Plot Density</td>
<td>⟨1⟩</td>
<td>Plot Density</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>Plot $v_x$</td>
<td>⟨!⟩</td>
<td>Plot log($v_x$)</td>
<td></td>
</tr>
<tr>
<td>(3)</td>
<td>Plot $v_y$</td>
<td>⟨@⟩</td>
<td>Plot log($v_y$)</td>
<td></td>
</tr>
<tr>
<td>(4)</td>
<td>Plot $v_z$</td>
<td>⟨#⟩</td>
<td>Plot log($v_z$)</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td>Plot Pressure</td>
<td>⟨$⟩</td>
<td>Plot log(Pressure)</td>
<td></td>
</tr>
<tr>
<td>(6)</td>
<td>Plot $B_x$</td>
<td>⟨%⟩</td>
<td>Plot Total $B$</td>
<td></td>
</tr>
<tr>
<td>(7)</td>
<td>Plot $B_y$</td>
<td>⟨^⟩</td>
<td>Plot $J \times B$</td>
<td></td>
</tr>
<tr>
<td>(8)</td>
<td>Plot $B_z$</td>
<td>⟨-⟩</td>
<td>Plot $\beta$</td>
<td></td>
</tr>
<tr>
<td>(9)</td>
<td>Plot div $B$</td>
<td>⟨0⟩</td>
<td>Plot log($\beta$)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Polygon Offset</th>
<th>Key</th>
<th>Synopsis</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨u⟩</td>
<td>Decrease Units</td>
<td>⟨U⟩</td>
</tr>
<tr>
<td>⟨m⟩</td>
<td>Decrease Factor</td>
<td>⟨M⟩</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Miscellaneous</th>
<th>Key</th>
<th>Synopsis</th>
</tr>
</thead>
<tbody>
<tr>
<td>⟨z⟩</td>
<td>Zoom In</td>
<td>⟨Z⟩</td>
</tr>
<tr>
<td>⟨d⟩</td>
<td>Window Snapshot</td>
<td>⟨c⟩</td>
</tr>
<tr>
<td>⟨d⟩</td>
<td>Save Screen Snapshot</td>
<td>⟨v⟩</td>
</tr>
<tr>
<td>⟨c⟩</td>
<td>Toggle Multisampling</td>
<td>⟨g⟩</td>
</tr>
<tr>
<td>⟨F1⟩</td>
<td>Toggle Selection Mode</td>
<td>⟨F2⟩</td>
</tr>
<tr>
<td>⟨F3⟩</td>
<td>Toggle Interpolation Mode</td>
<td>⟨F2⟩</td>
</tr>
</tbody>
</table>
Acknowledgments This work is supported by NASA Grant NAG5-10626 to University of Maryland and the NASA Goddard Space Flight Center's Computational Technologies Project.
Bibliography


VizFlow is a flexible visualization package that allows for visualization of two- and three-dimensional fluid dynamics and magneto-hydrodynamics (MHD) data that is adaptively-refined (AMR), multi-blocked, and multi-processor. It supports data in BATSRUS (U/Mich), AMRFCT (essentially PARAMESH (NASA/GSFC) format) and NEKTON (Argonne National Labs) formats. The data can be in Cartesian or Spherical coordinates. The package is downloadable from the Laboratory for Computation and Visualization, Institute for Physical Science and Technology (www.lcv.umd.edu).