Visualization in the Geosciences

Diplomarbeit von Christoph Moder

Oktober 2006

Betreuer:
Prof. Dr. Heiner Igel
Prof. Dr. Hans-Peter Bunge

Department für
Geo- und Umweltwissenschaften
Sektion Geophysik

Ludwig-Maximilians-Universität
München

Contents

1. Introduction

It is not enough to put thumbscrews on nature.
One has to understand when she testifies.
Arthur Schopenhauer

A great part of the knowledge that we have about the inside of the earth comes from direct and indirect observations. While the spectacular phenomena of volcanoes and earthquakes are known for thousands of years, the internal structure of the earth has not been known until the last century. New technologies like seismic measurements, gravimetry, ocean drilling and new methods in geodesy have shown us dynamic processes in the earth which were not known before.

Although these methods have revealed many internal structures of the earth, they could not explain them. Some behavior can be extrapolated from the conditions at the earth's surface, but since we cannot look into the earth and directly verify our assumptions, we have to make experiments. On a small scale this works very well; the behaviour of rocks and minerals is quite well understood. But experiments on a larger scale are much more difficult — one cannot create a mantle plume in a laboratory using the original environmental conditions, simply because these structures are far too large and develop too slow to be created by mankind. So, an earth scientist must not only control the experimental setup, but also the laws of nature — in other words, he must use computer simulations.

Another aspect is the compactness of the earth body; for example, a meteorologist can simulate a cyclone and compare his results with reality by taking measurements from an airplane flying through a hurricane. A geoscientist cannot do that; he cannot use intermediary results, he always has to simulate the whole interior up to the surface from where measurements are available. That leads to enormous amounts of data. This is the reason why it took about 100 years from Wiechert's first seismograph to simulations of wave propagation and mantle convection — not only the numerical methods had to be developed, but also necessary high-preformance computing hardware has not been available until these days (and still has to come).

But being able to build a model with correct properties does not imply being able to understand the results — let alone to be able to convey them in a comprehensible way for educational purposes, to laymen as well as to the media. In most cases, a numerical simulation provides only a heap of data, i.e. certain scalar or vector properties assigned to every grid point or grid cell. Yet the overview is still missing — being able to see interrelations means being able to handle the problem in an intuitive and descriptive way, to be able to look at it from different perspectives. Humans need to see structures! Since datasets are often multidimensional (e.g. vectors or tensors associated to every three-dimensional grid point), they have to be reduced so they can be represented in a three-dimensional space. Furthermore, it is much easier to see whether something exists than to see how it changes its behavior — although there are silicates from the earth's surface down to the lower mantle which only vary by their chemical composition and crystal lattice, we always think of sharp boundaries than of moreless smooth transitions. Hence, visualization means reduction of complex objects to visible structures and omission of properties that do not contribute to the aspect that should be shown.

Finally, intuition is also supported by interactivity. For computers, this requires the amount of data to be drastically reduced. For the image displayed on the screen, the internal composition of opaque objects does not matter; the hardware-accelerated 3D engines process only triangle meshes. So the challenge is: Turn your data into polygonal surfaces!

2. The Geowall

Although 3D projection systems have been existing for several years, their usage is still limited to a narrow area of application. This is mostly due to their high costs — the necessity to process big amounts of graphical data in real time together with non-standard input and output devices (like datagloves and a stereographic display) requires a lot of specialized high-performance hardware, which is expensive and in addition requires custom-made software.

But in the last years, off-the-shelf personal computers have become powerful enough to handle larger 3D sceneries. At the same time, graphics cards with dedicated 3D processors became available for PCs and have reached soon a complexity similar to the main processor, which has multiplied the 3D performance of PCs in a few years. And also video projectors which could provide fast frame rates in true color became available. With these ingredients — fast 3D accelerated PCs together with a projection device — immersive 3D projection has become possible with cheap standard hardware and standardized programming APIs[1]. Hereupon, a concept of a cheap alternative to a CAVE[2] setup has been developed, consisting of a standard PC, two projectors and polarizing glasses [Steinwand *et al.*, 2002]. It was named Geowall because it was ment for the immersive visualization especially of geoscientific data (see figure 2.1); the first Geowall has been installed in the Electronic Visualization Laboratory at the University of Chicago in the year 2001.

2.1. 3D Viewing

Three-dimensional viewing requires that the eyes see the object from slightly different points of view. This can be simulated by sending different images to the eyes — either with little screens or glasses with mirrors/prisms that show individual pictures to each eye, or by showing both pictures on the same screen with filtering glasses. The latter method is suited for a bigger audience because one single screen suffices for all people. Different filtering techniques are possible:

- Shutter glasses: If the two pictures are shown alternatingly, the spectators have to wear shutter glasses that are synchronized with the display. This technique is called "active stereo" since the glasses are electronic devices which actively change their behavior; this requires a screen with a refresh rate that is high enough to display both pictures in a flicker-free way.

[1]API = Application Programming Interface

[2]CAVE = CAVE Automatic Virtual Environment, a room with up to six screens on the walls, the floor and the ceiling

- Color filters: If the two pictures have different colors, cheap glasses with simple color filters are sufficient. Only monochromatic pictures are possible, but the screen type and the refresh rate are irrelevant.
- Polarizing filters: This technique combines full color pictures with cheap passive glasses, but requires a screen setup that can display pictures with polarized light (see section 2.2). Geowalls use this method.

All these filters are not perfect; there is always a little bit from the picture from the other eye to be seen. Normally this can be ignored, but if the pictures differ very much and if the contrast in the scene is very high, these ghost images can be bothersome.

The strength of the 3D effect is determined by the distance of the object. The closer the object comes, the more impressive is its 3D view — but also ghost images become noticeable, and because the focal plane remains at the screen, viewing becomes exhausting. This means that it can be necessary to change the virtual eye distance; it has to be reduced if details of the object are viewed from a close (virtual) distance, and it can be enlarged to increase the 3D effect of objects further away. See also section 4.1.3 how to accomplish this with the VTK[3] toolkit.

Common to all 3D projection methods is that the spectators get only a 3D view of the scene, but the picture itself is still flat. This demands for interactivity — if the spectator cannot change his position relative to the object, the displayed object must be free to move. Hence, static 3D picture viewers are not very useful; interactive software is necessary, or at least precomputed animations.

2.2. Hardware

The following components are required for a Geowall:

- *PC:* There are no special requirements except a high enough performance, i.e. enough RAM (more than 1 GB) and a fast processor. Any operating system can be used, but most software is available for Linux and Windows.
- *Graphics card:* It must be a "dual head graphics card", i.e. have connectors for two screens. A fast 3D chipset is not mandatory, but absolutely desirable. For non-Windows systems, a good driver support is essential, so most Linux-based Geowalls use nVidia graphics cards. If the so-called "clone mode" (see section 2.3) should be used, a quad-buffered graphics card is necessary; in our case, it is a nVidia Quadro FX 1400.
- *Projectors:* Since polarizing filters are used, the projectors must not emit polarized light — which LCD projectors do. So only DLP[4] projectors which are based on an array of tiny mirrors can be used. Here, we are using NEC LT245 projectors.

[3]See section 3.1

[4]DLP$^{\text{TM}}$= Digital Light Processing; the brightness of the pixels is controlled by pulse-width modulated mirror tilting frequency

- *Polarizing filters:* Either linear or circular polarized filters can be used; the latter are more expensive, but offer the advantage that the spectators can tilt their heads without losing the 3D effect. Filters made of glass are very expensive; in contrast, plastic film filters are sensitive to the heat of the projectors, so they should be enclosed between thin glass panels.
- *Polarizing glasses* for the audience; cheap cardboard glasses are sufficient. The filters used in the glasses must match the filters at the projectors, so both have to be either linear or circular polarized.
- *Metallic screen:* Reflection on most screens polarizes the light, so the 3D effect is lost. To preserve the original polarization, the screen must have a metallic surface. Special 3D screens are quite expensive; but also other metal-coated fabric can be used[5]. Metal plates (e.g. aluminum) can also be used, but the highlights from the projectors on the surface are disturbing.

2.3. Screen Setup

The two images must be output on the two projectors. This can be achieved with two different configurations of the graphics driver (the terms are the same as the nVidia driver uses):

- *"RightOf":* This mode means that the two projectors or monitors are arranged side-by-side so that there is a big virtual desktop that stretches across the two monitors. In this case, the 3D software must display the two images also side-by-side. When the window containing these images is enlarged to the full desktop size, each of the images is displayed on one projector.
  This technique has the advantage that it works with every dual-head graphics card, and also the software does not need special libraries to draw the two images.
- *"Clone":* In this mode, the two projectors show the same image. But using the GLX library (which provides OpenGL functions under the X Window System), special 3D commands create different views of the 3D object on the two screens.
  This has the advantage that also the normal 2D elements are visible since they are displayed on both projectors. But only the more expensive graphics cards with quad buffering (i.e. double buffering, which allows a flicker-free picture on the screen while the next picture is being created, for both screens) support this mode.
  Because the use of OpenGL is widespread, there are several 3D applications available using this graphics mode[6]; additionally, if OpenGL-based software uses other 3D modes (e.g. red-blue), it is often easy to add this active stereo mode — for example for the game *Nexuiz* only 22 lines of code are required[7].

Under Linux, these modes are set in the file `/etc/X11/XF86Config-4` or `/etc/X11/xorg.conf`, respectively; see section 5.1.

[5]Here, the silver screen fabric from Vision24 has been used, see `http://www.vision24.de/ProfiScreen/screens/Meterware.htm`.

[6]See `http://www.reald.com/scientific/stereo_ready_software.asp`

[7]See `http://www.alientrap.org/forum/viewtopic.php?t=864`.

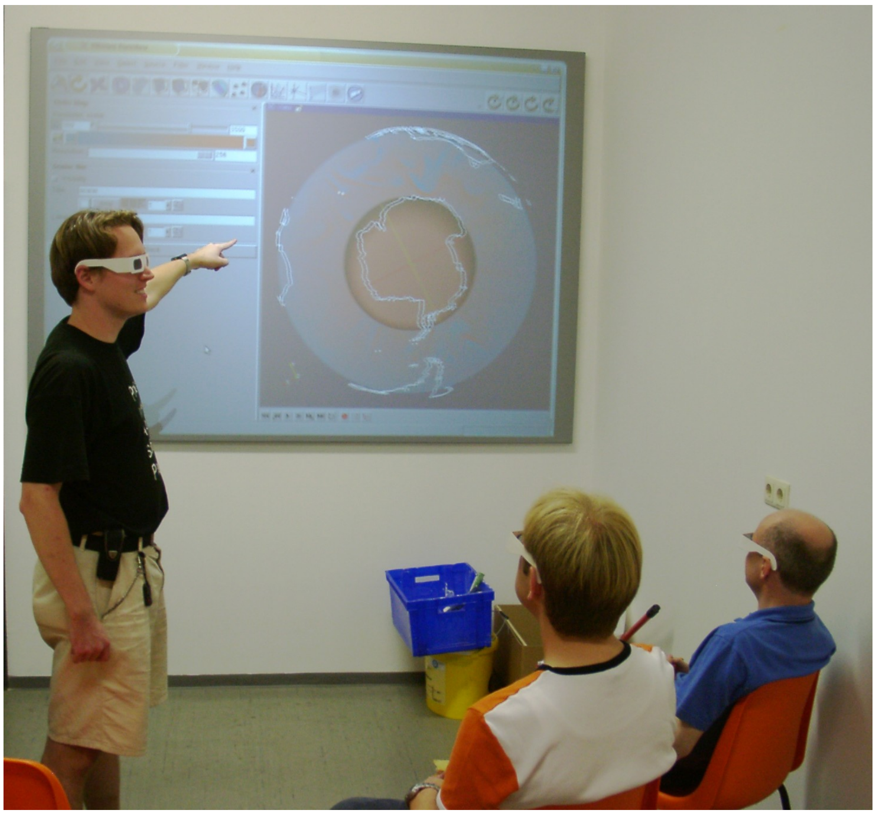


Figure 2.1: Geowall showing a 3D cross-section of the earth, overlaid with coast lines.

3. VTK and Paraview

*The commonality between science and art is in trying to see profoundly —
to develop strategies of seeing and showing.*
Edward Tufte

In every complex electronic device, not only the hardware, but also the software determines its capabilities. This applies also for the Geowall. The usual applications — stereo image viewers, stereo movie players and VRML[1] viewers — are rather mere playback devices. But this tool can be much more powerful if the data can also be edited interactively. With the Visualization Toolkit (VTK) and Paraview there is a very mighty and versatile set of open-source applications which can be used on the Geowall.

3.1. The Visualization Toolkit (VTK)

VTK is a high-level library for data processing and visualization, providing sophisticated functions like contouring, cutting, triangulation and mesh smoothing. Its strengths are particularly modularity and portability — it is available both for Linux and Windows, can be used with a variety of programming languages and offers not only isolated functions, but a complete workflow starting from raw data until the rendered image, optionally fully interactive. VTK can be used either as a classical library for C++ programs as well as with Java or scripting languages[2] like Tcl or Python — which gives a very high-level access to the processing functions, with a few lines of a scripting language extensive data processing can be done ("software leverage"). Because of its modular object-oriented layout, the code is reusable and layers in the visualization process (like the rendering implementation) can easily be exchanged, for a tailor-made trade-off between generality and efficiency. Despite its universal concept, its performance is quite high by using techniques like OpenGL[3] for the access to hardware-accelerated rendering and multithreading and MPI[4] for parallel processing.

[1]VRML = Virtual Reality Markup Language, a plain-text description language for three-dimensional sceneries

[2]There is even a wrapper code generator which simplifies the development of interface code for new programming languages; see [Schroeder *et al.*, 1996].

[3]OpenGL: a platform-independent programming interface for 3D graphics

[4]MPI = Message Passing Interface, see section 4.2

3.1.1. File Formats

VTK has several proprietary file formats which support different grid types (structured, unstructured etc.) as well as different ways of storing the data (ASCII, raw binary, compressed binary). These formats are documented in the manual [Kitware Inc., 2004] and also online [Kitware, 2006].

Grid Types

There are several subformats for every type of grid; all of them use a right-handed cartesian coordinate system.

- The most general grid is the unstructured-grid format where every grid point is given by its coordinates and additionally the connectivity between the points; for every cell the constituent grid points and its shape, like tetrahedral or cubic, are given.
- A polygonal grid is similar, but has 2D cells (polygons) instead of 3D cells (polyhedra).
- In a structured grid every grid point has six neighbors which are given by the point order, so there is no explicit information about the connectivity needed.
- The cells of a rectilinear grid are cuboids, so only the intervals on the axes have to be given instead of the point coordinates.
- A grid of structured points has cubic cells, so only the grid spacing has to be given, which makes this format very RAM-friendly.

File Format Types

There are two families of file formats. The old VTK format uses plaintext headers between the data sections which can be written in ASCII or binary; the filename extension is always `*.vtk`, independent from the grid type. The new XML-based[5] file formats enclose the data sections in XML tags; their filename extension depends on the grid type, e.g. `*.vtu` for unstructured grids or `*.vtp` for polygonal grids.

The following file format types are available:

- VTK ASCII: Between the section headers the data is written in ASCII; different number formats are allowed (e.g. `202.3` or `2.023E+2`), the separators can be spaces, tabs or line breaks.
- VTK binary: Similar to the ASCII format, but after the section headers only the raw binary data is allowed — no spaces, no additional line breaks etc. The data types are the usual 32-bit data types (big endian) known from C programs, e.g. *float* is a 32-bit floating point number, and *int* is a 32-bit integer variable.

[5]XML = Extensible Markup Language

- XML ASCII: Here, the section headers are replaced by XML tags. But also further minor changes are possible; e.g. each line in the `CELLS` section of a VTK unstructured grid file is prepended by the number of values that follow, whereas in the XML format there is an extra data array which contains the offsets. If a cell definition in the VTK format looks like
  ```
  8 0 9 3 12 1 10 4 13,
  ```
  its XML counterpart is
  ```
  0 9 3 12 1 10 4 13
  ```
  with an offsets array like
  ```
  8 16 24 32 40 ...
  ```

- XML binary inline: Since XML actually does not allow certain special characters (as they may appear in binary data), the data should be base64-encoded[6]. The length of the data is prepended, so a pseudocode to write base64-encoded data would look like this:
  ```
  int32 = length( data );
  output = base64-encode( int32 )+ base64-encode( data );
  ```

- XML binary appended: The data can also be written as a coherent chunk at the end of the file; the section headers then contain byte offset values into the data section.

- XML binary compressed inline/appended: The binary data can also be compressed. Because processors are several magnitudes faster than disk I/O, reading and writing can thus be accelerated since the amount of data is reduced with negligible processing time. Like in the uncompressed case, a header is prepended, consisting of four 32-bit variables; the first two of them seem to have fixed values. A pseudocode would look like this:
  ```
  int32[1] = 1;
  int32[2] = 32768;
  int32[3] = length(data);
  zdata = zlib-deflate( data );
  int32[4] = length( zdata );
  output = base64-encode( int32 )+ base64-encode( zdata );
  ```

- XML parallel: This file is just a wrapper that lists all the files that belong to this dataset. If e.g. a dataset has been split up into 10 unstructured grid files, this wrapper file has the extension `*.pvtu` and lists the filenames of the 10 `*.vtu` files. With this file format, a bunch of data files can be loaded at once, and if the software is run on multiple computers, each process can read its data independently from the others at the same time.

Although the user is encouraged to use VTK's library functions to write the data files, this is not really necessary because especially the old legacy VTK file formats are simple enough to be written by any application. ASCII has the advantage that it is platform-independent (no problems with sizes of the variables and endianness), whereas binary files load much faster because their content can be copied directly into the RAM without any parsing.

[6]base64: a code that maps three 8 bit binary data values on four 6 bit numbers which can be represented by normal ASCII characters, without spaces and special characters; this is used when binary data must be embedded in plaintext files, especially in emails and together with markup languages like XML

Listing 3.1: Example for an ASCII VTK file describing a cube. From top to bottom: the 27 vectors of the coordinate points, the 8 cube-shaped cells (in each line there is the number of the cell points (= 8) and then the indices of the corresponding point coordinates), the 8 cell types (type 11 = cubic), and then the datasets — first 8 vectors for each cell, then 27 scalars for each grid point

```
# vtk DataFile Version 3.0
Demo: Cube
ASCII
DATASET UNSTRUCTURED_GRID
POINTS 27 float
0 0 0
0 0 1
0 0 2
0 1 0
0 1 1
0 1 2
0 2 0
0 2 1
0 2 2
1 0 0
1 0 1
1 0 2
1 1 0
1 1 1
1 1 2
1 2 0
1 2 1
1 2 2
2 0 0
2 0 1
2 0 2
2 1 0
2 1 1
2 1 2
2 2 0
2 2 1
2 2 2

CELLS 8 72
8 0 9 3 12 1 10 4 13
8 1 10 4 13 2 11 5 14
8 3 12 6 15 4 13 7 16
8 4 13 7 16 5 14 8 17
8 9 18 12 21 10 19 13 22
8 10 19 13 22 11 20 14 23
8 12 21 15 24 13 22 16 25
8 13 22 16 25 14 23 17 26

CELL_TYPES 8
11
11
11
11
11
11
11
11

```

```
CELL_DATA 8
VECTORS mycellvectors float
0 -2 -1
0 -2 1
0 2 -1
-1 2 1
2 -1 -2
1 -2 1
2 2 -1
2 2 1

POINT_DATA 27
SCALARS mypointscalars float
LOOKUP_TABLE mytable
-0.4 -0.4 -0.4 -0.4 -1 -0.4 -0.4 -0.4 -0.4
0.4 1 0.4 1 0.4 1 0.4 1 0.4
0.4 0.4 0.4 0.4 1 0.4 0.4 0.4 0.4

LOOKUP_TABLE mytable 4
   6.7397900e-02   7.3764400e-02   9.5253700e-01   1.0000000e+00
   7.9526300e-02   8.7038400e-02   9.3611300e-01   1.0000000e+00
   9.3329400e-02   1.0214500e-01   9.2136300e-01   1.0000000e+00
   1.0893600e-01   1.1922600e-01   9.0841600e-01   1.0000000e+00
```

If binary data should be written, the tools from Seismic Unix [Stockwell and Cohen, 2002] can be helpful; e.g. *swapbytes* (to convert from little endian to big endian) or *ftnstrip* (to remove the extra bytes from Fortran's `UNFORMATTED` file type).

Converting Between File Formats

In practice, it is the easiest to stick to a simple legacy file format (which is easier to write) and convert it to a XML-based format (which is needed in parallel formats) using a VTK script like this:

Listing 3.2: Converting an ASCII VTK file to a compressed binary XML VTU file

```
#!/usr/bin/env tclsh

package require vtk

vtkUnstructuredGridReader reader
        reader SetFileName [lindex $argv 0]

vtkZLibDataCompressor compressor

vtkXMLUnstructuredGridWriter writer
        writer SetInput [reader GetOutput]
        writer SetFileName [lindex $argv 1]
        writer SetDataModeToAppended
        writer SetCompressor compressor
        writer EncodeAppendedDataOn
        writer Write

exit
```

3.1.2. The Visualization Pipeline

VTK is a data-flow driven system; the basic concept is a user-defined pipeline which describes the data flow from the input files to the screen. The program sequence results from the desired data flow, so the user does not have to worry about the program logic — if there are several parallel data pipelines, then the processing is automatically parallelized (similar to a Unix shell script or a LabView program). Since VTK is strictly object-oriented, every processing step is represented by a self-contained object which takes its input from the previous element in the pipeline and passes its output to the next one. In Tcl, this looks like that:

```
        pipelineObject SetInput [previousObject GetOutput]
```

At the end of the pipeline there is a data source or a data sink, respectively. A data source is usually a reader object (e.g. *vtkUnstructuredGridReader*) that reads in the file with the given name, and a data sink can be a writer object which outputs the data to a file (e.g. *vtkXMLPolyDataWriter*). The display on the screen is controlled by several objects: A *vtkRenderWindow* object provides the screen space where one or more vtkRenderer objects can draw their images; *vtkLight* and *vtkCamera* control the illumination and view parameters, and a *vtkActor* object represents one item on the screen — together with *vtkProperty* (which defines its surface properties) and *vtkMapper* (which represents the geometric shape of the object). The latter is usually the data sink of a processing pipeline and maps properties of the dataset (like data arrays) to visible features like colors by using e.g. lookup tables. Between the source and the writer or mapper there can be an arbitrary number of filters that modify the data.

A visualization pipeline is not necessarily linear. Several data objects can be displayed at the same time, which can originate from the same data source or the same filtering object. Likewise, there are filters that combine several inputs into one single pipeline. Even loops are possible (i.e. a filter takes its output as input), but are evaluated only once (except forced by an explicit update command). The data flow is always from source to sink, in combination with a lazy evaluation algorithm: Only if a data sink requests an update, the data in the pipeline is recomputed. Starting from the sink, the update requests are handed on to the previous elements in the pipeline until the necessary source data is available; the newly computed data travels then downstream through the pipeline until the requesting sink is reached.

3.1.3. Processing Optimizations

The pipeline concept gains its flexibility especially at the cost of RAM efficiency — normally, every element in the pipeline has its own copy of the data. If there is a lack of memory, the pipeline can be shortened by saving intermediate steps into new files and unloading the files with the previous processing steps. Also the order of the pipeline elements plays a decisive role; filters with a strong reduction of the amount of data should come first. For example, if a cross-section of an isosurface should be generated, it is favorable to create the isosurface at first, because in the second step only a small amount of data has to be cut through.

Also the processing can be tuned. With the option ImmediateModeRendering (also available on the Paraview GUI, menu item “View”/“3D View Properties”), the creation of OpenGL display lists, which require an additional copy of the data, can be disabled. For VTK, there are furthermore the options `ReleaseDataFlagOn` or `GlobalReleaseDataFlagOn` which force the filters in the pipeline to discard their intermediate results after communicating them to the next filter.

Finally, polygonal objects can be optimized by converting them to triangles (`vtkTriangleFilter`) and then to triangle strips (`vtkStripper`), which are more efficient to render.

3.2. Paraview

Paraview is a graphical visualization software based on the VTK. That means, the underlying algorithms, the file formats and the rendering capabilities are identical. However, Paraview is not just a GUI for the VTK, like MayaVi. The basic idea is that a maximum of performance can be achieved if every available hardware can be used and always the best available platform can be chosen — without need for expensive tailor-made computers. With a maximum of portability and scalability the software can be fitted to almost every available hardware configuration. Unlike other programs like OpenDX or SCIRun, there is not a "central executive", but almost everything in Paraview can be parallelized — on shared-memory machines with pthreads or sprocs, on distributed memory architectures with MPI.

The following cases of parallelization can be distinguished [Ahrens *et al.*, 2000]:

- Task parallelism means that independent processing pipelines can work at the same time.
- Pipeline parallelism means that different steps of one processing pipeline can work at different processing stages of the same dataset. This is favorable especially if the different processing steps require different resources (input/output or CPU time).
- Data parallelism means that the data is partitioned, and several identical processing pipelines work at different pieces of the dataset. A load-balancing algorithmus breaks the dataset in similar pieces; each module in the pipeline is told by the following module which output it has to generate and thus determines what input it requires. Since all processes do the same task on different data, they all need the same types of resources, but can run quite independently ($\Rightarrow$ few communication necessary), so they should run on different machines.

Especially data parallelism makes only sense if the processing pipelines use different computers; therefore, an additional communication layer was necessary. The VTK modules have been equipped with the ability to write their data into strings that are passed on by client server streaming modules. By adding information about the data type and byte ordering, these streaming modules offer a cross-platform way of communication between Paraview components.

For every copy of a VTK module that is running on a remote machine, the client maintains a proxy object that stores its state. These proxies (`vtkSMProxy` and its subclasses) are coordinated by the server manager. This additional abstraction layer allows also a client-server architecture with a separation between data and their graphical representation, or accordingly CPU intensive data processing and GPU[7] intensive rendering.

This flexibility has the drawback that VTK commands cannot be issued directly, but only via Server Manager objects, if available:

```
        [[$Application GetMainWindow] GetCurrentPVSource] GetPVColorMap]
            SetScalarRange 0 0.5 ]
```

The mapping between VTK objects and server manager objects is done via XML configuration files, as well as their entries in the user interface:

Listing 3.3: Mapping between VTK filter objects and server manager objects (taken from *Servers/ServerManager/Resources/filters.xml*)

```
   <SourceProxy name="ExtractEdges" class="vtkExtractEdges">
     <InputProperty
        name="Input"
        command="SetInput">
           <ProxyGroupDomain name="groups">
             <Group name="sources"/>
             <Group name="filters"/>
           </ProxyGroupDomain>
           <DataTypeDomain name="input_type">
             <DataType value="vtkDataSet"/>
           </DataTypeDomain>
      </InputProperty>
   <!-- End ExtractEdges -->
   </SourceProxy>
```

Listing 3.4: Configuration of the user interface (taken from *GUI/Client/Resources/Filters.xml*)

```
  <Module name="ExtractEdges"
          menu_name="Extract Edges"
          root_name="ExtractEdges"
          module_type="Filter"
          long_help="Extract edges of 2-d and 3-d cells as lines."
          short_help="Covert data to wireframe.">
    <Filter class="vtkExtractEdges">
      <Input name="Input"
             class="vtkDataSet"/>
    </Filter>
    <InputMenu trace_name="Input" label="Input" property="Input"
               help="Set the input to this filter."
               input_name="Input"/>
    <Documentation>
The Extract Edges filter produces a wireframe version of the input data set by
    extracting all the edges of the data set's cells as lines. The Input menu
    allows the user to select the data set to which this filter will be applied.
    This filter operates on any type of data set and produces polygonal output.
    </Documentation>
  </Module>
```

[7]GPU = Graphics Processing Unit, a highly parallelized graphics processor on the graphics card

3.3. Extensions

There are several interesting third-party tools and extensions for the use with VTK available; the following extensions have not been tested; for the sake of completeness, they are mentioned.

- *Matlab* can also use VTK by compiling VTK programs with the *mex* compiler. An example is given at [Szczerba, 2006]. With the tool *vtkAutomex* the creation of such C++ programs can be automated [Ghosh, 2004]. There are even wrappers that make VTK accessible from Matlab scripts; the package is called *Octaviz* [Octaviz].
- *GRASS*, an open-source GIS system, has some scripts included which can export raster and vector maps into the VTK format [GRASS, 2006].
- *vtkCSCSNetCDF* is a reader for netCDF files that can be compiled into Paraview [Biddiscombe, 2006].
- *vtkGeography* is a series of VTK classes that provide mainly functions for reading and writing GIS file formats [Flanagin, 2005].
- *vtkRIBExporter* exports a VTK scene for rendering with a RenderMan compatible renderer [Lorensen, 2000].
- *vtkFVR* is a series of VTK classes for virtual reality environments which use the FreeVR library [Sites, 2003].
- *vtkActorToPF* renders using the SGI Performer library [Rajlich, 2006].
- *GL2PS* is a library which peeks into the OpenGL feedback buffer and translates its content into Postscript or other vector formats [Geuzaine, 2006].

4. Building Paraview with Support for MPI and Geowall

In theory, there is no difference between theory and practice.
But, in practice, there is.
JAN L. A. VAN DE SNEPSCHEUT

Although there are ready-made binary packages of Paraview for several operating systems, one has to compile it from source code to use features like MPI, Mesa offscreen rendering and support for true color stereographic viewing.

Note: The mentioned modifications and configuration options refer to Paraview version 2.4.4.

4.1. Adding True Color Stereo Support

The underlying VTK framework does have support for stereographic output, i.e. stereographic views can be generated from spatial objects and these views can be output for different stereo viewing systems. The default setting is the red-blue stereo mode, and since the Paraview GUI[1] does not offer to change this, the source code must be modified to allow one of the other stereo modes.

4.1.1. Activating GLX Stereo Mode

VTK supports the stereo functions of the GLX library which uses the Clone mode (see section 2.3) of a quad buffered dual-head graphics card. This mode is called `VTK_STEREO_CRYSTAL_EYES` since it was meant for CrystalEyes® LCD shutter glasses. To activate this mode, the initialization of the stereo mode must be changed. This is done in the file *VTK/Rendering/vtkRenderWindow.cxx* in the function `vtkRenderWindow::vtkRenderWindow()`:

```
        this->StereoType = VTK_STEREO_CRYSTAL_EYES;
```

[1]GUI = Graphical User Interface

4.1.2. Adding a Side-by-Side Stereo Mode

If the graphics card does not support quad buffering, the side-by-side stereo mode (see section 2.3) can be used. This mode is not natively supported by VTK, but it requires only minor additions to the source code (contributed by Thomas Hansen).

Listing 4.1: Modifications to *VTK/Rendering/vtkRenderWindow.h*

```
#define VTK_STEREO_ANAGLYPH      7
#define VTK_STEREO_SVEN          8
```

```
  void SetStereoTypeToAnaglyph()
    {this->SetStereoType(VTK_STEREO_ANAGLYPH);};
  void SetStereoTypeToSVEN()
    {this->SetStereoType(VTK_STEREO_SVEN);};
```

```
    case VTK_STEREO_ANAGLYPH:
      return (char *)"Anaglyph";
    case VTK_STEREO_SVEN:
      return (char *)"SVENDisplay";
    default:
```

The actual stereo view is generated in the function
`vtkRenderWindow::StereoRenderComplete(void).`
As one can see, a new input buffer is created and filled from the two stereo buffers. Only every other pixel is copied, so both stereo frames fit into the new frame:

Listing 4.2: Modifications to *VTK/Rendering/vtkRenderWindow.cxx*

```
      case VTK_STEREO_INTERLACED:
        this->StereoStatus = 1;
        break;
      case VTK_STEREO_SVEN:
        this->StereoStatus = 1;
      }
```

```
      case VTK_STEREO_INTERLACED:
        this->StereoStatus = 0;
        break;
      case VTK_STEREO_SVEN:
        this->StereoStatus = 0;
      }
```

```
      (this->StereoType == VTK_STEREO_DRESDEN) ||
      (this->StereoType == VTK_STEREO_ANAGLYPH) ||
      (this->StereoType == VTK_STEREO_SVEN))
    {
```

```

// SVEN STEREO ////////////////////////////////////////////////////////////////
// <thomas.hansen@wartburg.edu> ///////////////////////////////////////////////
//
    case VTK_STEREO_SVEN:
      {
      unsigned char *buff;
      unsigned char *p1, *p2, *p3;
      unsigned char* result;
      int *size;
      int halfLength;  //hlaf thelength, each eye gets half of teh window
      int x,y;
      int res;

      // get the size
      size = this->GetSize();
      halfLength = size[0]/2;

     // get the data
      buff = this->GetPixelData(0,0,size[0]-1,size[1]-1,!this->DoubleBuffer);
      p1 = this->StereoBuffer;
      p2 = buff;

      // allocate the result
      result = new unsigned char [size[0]*size[1]*3];
      if (!result)
        {
        vtkErrorMacro(<<"Couldn't allocate memory for SVEN stereo.");
        return;
        }
      p3 = result;


      // now put the two images next to each other
      for (y = 0; y < size[1]; y ++)
        {
        //Needed in case there is an odd number of pixel columns.
        //Teh skipped line prevents some very odd results
        if(size[0]%2!=0){
          *p3++ = 0;
          *p3++ = 0;
          *p3++ = 0; p1 += 3; p2 += 3;
        }

        //left eye on the left half
        p1 +=  halfLength/2 * 3;
        for (x = 0; x < halfLength; x++)
          {
          *p3++ = *p1++;
          *p3++ = *p1++;
          *p3++ = *p1++;
          }
        p1 += (halfLength - halfLength/2) *3;

        //righ eye on the right half
        p2 +=  halfLength/2 * 3;
        for (x = 0; x < halfLength; x++)
```

```
            {
            *p3++ = *p2++;
            *p3++ = *p2++;
            *p3++ = *p2++;
            }
          p2 += (halfLength - halfLength/2) *3;
          }

        this->ResultFrame = result;
        delete [] this->StereoBuffer;
        this->StereoBuffer = NULL;
        delete [] buff;
        }
        break;
// SVEN STEREO /////////////////////////////////////////////////////////////////////
      }
```

4.1.3. Changing the Virtual Eye Angle

VTK seems to use the "toe-in" stereo mode [Bourke, 2002] where the two cameras do not look asymmetrically at the same virtual plane, but on two different planes which are perpendicular to the cameras (which results in vertical parallax at the corners). The virtual eye distance can then be expressed by the angle between the view axes. By default, this virtual eye angle in VTK is set to 2°. This is a reasonable value for a general 3D view of the object, but too much if one zooms in to see some details enlarged. The exaggerated 3D effect can be weakened by changing the variable `EyeAngle` in the function `vtkCamera::vtkCamera`, for example to 0.5 instead of 2.0:

Listing 4.3: Setting a smaller view angle in *VTK/Rendering/vtkCamera.cxx*

```
  this->EyeAngle = 0.5;
  this->Stereo = 0;
  this->LeftEye = 1;
```

4.2. MPI Support

MPI stands for *Message Passing Interface*, which is an API for parallelized programs running on distributed-memory machines. This interface offers communication routines for data exchange between the program instances; but since it defines only the interface, not the implementation, the startup script that is used must match the MPI library used by the program.

Multiprocessing does make sense for visualization tasks — mainly not because of the computing power (in real-time applications data has to be transmitted frequently, so the network bandwidth likely becomes a bottleneck), but because of the memory requirements; so Paraview and also the underlying VTK are prepared to use MPI. However, since there are many different implementations available, the binaries are shipped without MPI support. For using it, Paraview has to be compiled from source code.

There are several free MPI implementations like MPICH (tested version: 1.2.7), MPICH2 (tested version: 1.0.3) and LAM-MPI (tested version: 7.1.1). The latter did not work together with Paraview 2.4.4.

4.2.1. MPICH

Compiling MPICH works flawlessly:

```
$ ./configure
$ make
# make install
```

However, it has been recommended to increase the maximum packet size in the source code (from 2^{28} = 256 MB to 2^{30} = 1 GB):

Listing 4.4: Increasing the maximum packet size in *mpid/ch_p4/p4/include/p4_sr.h*

```
#define P4_MAX_MSGLEN (1<<30)  /* Used in free_p4_msg as sanity check
                                  increase as desired */
```

4.2.2. MPICH2

MPICH2 sounds like the successor of MPICH which it indeed claims to be, but the source code has been rewritten from scratch. One difference is that MPICH2 uses a process manager called *mpd* which must be loaded before running *mpiexec*. Compiling and installing MPICH2 is similar to MPICH; however, linking the library against a C++ program (like Paraview) fails because of a duplicate definition of the `SEEK_XXX` constants. This can be avoided with the compiler flag`-DMPICH_IGNORE_CXX_SEEK` (to be set in Cmake) [Gropp *et al.*].

4.3. Offscreen Rendering

Remote rendering processes can either open X windows to draw their images, or they can use “offscreen rendering” — which means that an image is rendered without displaying it in a window; only few OpenGL drivers are capable of this. One exception is the Mesa library, a free OpenGL implementation. Since it does not use any hardware acceleration, providing offscreen rendering is hassle-free.

It is even possible to use both a hardware-accelerated OpenGL driver (for visible rendering) and the Mesa library (for offscreen rendering) in the same program. To be able to use both implementations of OpenGL, the libraries must not have the same function names. This can be achieved with a so-called “mangled Mesa library”; the Mesa library does have an option which changes all function names such that they do not interfere with a second OpenGL library. It can be done like this:

```
$ tar xfvj MesaLib-6.5.tar.bz2
$ cd MesaLib-6.5
$ vi configs/default
        CFLAGS = -DUSE_MGL_NAMESPACE -O
        CXXFLAGS = -DUSE_MGL_NAMESPACE -O
        GL_LIB = MGL
        GLU_LIB = MGLU
        GLUT_LIB = Mglut
        GLW_LIB = MGLw
        OSMESA_LIB = MOSMesa
$ make linux-x86-static
```

Paraview offers configuration options to be compiled with a mangled Mesa library — but my attempts of compiling it have not been successful. However, one can create two versions which differ in the OpenGL library they use. Since there are computers without graphics hardware (like cluster nodes), a Paraview version using the (unmodified) Mesa library (instead of a graphics driver) must be available anyway.

4.4. Compiling

Paraview does not use the widespread GNU *autoconf/automake* mechanism, but uses *CMake* which offers an interactive text user interface (*ccmake*). It is suggested to execute the build process in a separate directory which allows to build several versions independently from the same source code directory.

Listing 4.5: Unpacking and compiling Paraview

```
$ tar xfvz paraview-2.4.4.tar.gz
$ mkdir paraview-2.4.4-build
$ cd paraview-2.4.4-build
$ ccmake ../paraview-2.4.4
$ make
```

ccmake offers a plethora of configuration options for Paraview. Although ccmake can detect most pathnames automatically, it is strongly recommended to verify them, since there are often several versions of libraries and include files available on one system (e.g. the correct include files for the nVidia graphics card are not in */usr/include*, but in */usr/share/nvidia/include*). And I do recommend using static libraries because I have experienced incompatibilities with the installed MPICH shared library, and on some machines (esp. the cluster nodes) certain libraries were not available at all.

The following settings have proved to be useful:

Listing 4.6: *ccmake* options

```
# general
VTK_USE_64BIT_IDS
PARAVIEW_WRAP_PYTHON
PARAVIEW_EXPERIMENTAL_USER
```

```
# when MPI is desired
VTK_USE_MPI
VTK_MAX_NUMPROCS 130
MPI_INCLUDE_PATH /home/username/local/src/mpich-1.2.7/lib/libmpich.a
MPI_LIBRARY /home/username/local/src/mpich-1.2.7/include
MPI_EXTRA_LIBRARY /home/username/local/src/mpich-1.2.7/lib/libmpich.a
```

After pressing *c* for “configure” and then *g* for “generate”, the source code is copied to the destination directory and suitable makefiles are created:

```
$ make
$ make install
```

5. Running Paraview in 3D and with Large Datasets

The problem with troubleshooting is that the trouble shoots back.
ANONYMOUS

5.1. Setting the Stereo Mode

In general, the Paraview GUI (*paraview* or *pvclient*, respectively) is launched in stereo mode with the following command line option:

```
$ paraview --stereo &
```

As described above, the stereo mode cannot be chosen at runtime, contrary to other VTK based applications like MayaVi. And since there does not seem to be a possibility to do it via Tcl scripting at the built-in command line, the only solution is to compile several versions with different stereo modes, if needed.

The default red-blue stereo mode runs on every computer and does not require any special setting. The true color stereo modes (see section 2.2), in contrary, require a dual-head graphics card and the following settings:

Listing 5.1: Settings in */etc/X11/XF86Config-4*

```
Section "Device"
        Identifier      "nVidia Corporation NV41GL [Quadro FX 1400]"
        Driver          "nvidia"
        Option "NvAGP" "2"
        Option "NoLogo" "true"
        Option "TwinView" "true"
        Option "SecondMonitorHorizSync" "10-100"
        Option "SecondMonitorVertRefresh" "50-120"
        Option "MetaModes" "1024x768,1024x768"

        # Rightof Mode
        # Option "TwinViewOrientation" "RightOf"

        # Clone Mode
        Option "TwinViewOrientation" "Clone"

        Option "Stereo" "4"
EndSection
```

5.2. MPI Usage

Numerical simulations use MPI because of the computing power, whereas for visualization tasks the memory is the main problem, since the coordinate grid and the connectivity information are not generated sequentially, but have to be kept in memory together with the data. One way to handle large datasets is spreading them over several computers using MPI-enabled Paraview.

MPI programs compiled with the MPICH library are started with the command *mpirun*:

```
$ mpirun -machinefile machines.txt -np 32 paraview
```

The number of processes is given by the parameter`-np` . If there are at least this many entries in the machinefile, every process has its own machine. If not, then there are several processes per computer.

MPICH2 uses a slightly different approach; the process management is now separated from the interprocess communication, thus giving more flexibility and scalability, but a process manager (called *mpd*) has to be started on every node [Gropp *et al.*].

```
$ mpdboot --file=machinefile.txt        # starts mpd on all nodes
$ mpiexec -n 32 paraview
$ mpdallexit            # after the execution, remove all instances of mpd
```

Because SSH[1] is used to exchange data, it is reasonable to generate public/private keys to avoid having to type the password for each process. This can be done the following way:

```
user@local_machine$ ssh-keygen          # without passphrase
user@local_machine$ scp identity.pub user@remote_machine:
user@remote_machine$ mkdir .ssh
user@remote_machine$ cat identity.pub >> .ssh/authorized_keys
user@remote_machine$ chmod 600 .ssh/authorized_keys
user@remote_machine$ rm -f identity.pub
```

Note: Not only the SSH public key, but also the program binaries (including all necessary libraries!) and the data files have to be available on each MPI node, either via a shared file system or copied to every node.

In a composite rendering setup there is an additional difficulty. When an image is computed, it has to be displayed in a window; for this, the server nodes have to be configured such that the rendering software can open X windows, or that X connections are possible over the network. The usual answer to the latter is something like `xhost +` on the client which should allow X connections from the server nodes, but host-based authentication is generally considered harmful (hence modern Linux distributions prevent all unencrypted X connections with `startx -nolisten tcp`); a better approach is the usage of `xauth`. Also X forwarding via *ssh* is possible, yet quite slow. I can be accomplished with the following settings in `$HOME/.ssh/config`:

```
HOST *
  ForwardX11 yes
```

[1]SSH = Secure SHell

5.3. Client-Server Mode

One outstanding feature of Paraview is its client-server architecture. That means, there is not necessarily one monolithic program doing all the processing tasks, but it can be distributed over several machines. The three components in the client-server model are:

- the data server: stores the raw data and executes the filtering algorithms
- the render server: takes the polygonal surfaces and renders an image
- the client: displays the image and provides the user interface

This has the advantage that for each task specialized machines can be used. For the raw data, machines with a lot of RAM and fast processors are needed, but do not need any graphics hardware. Because the amount of surface data is usually much smaller than the original dataset [Cedilnik *et al.*, 2006], less render servers than data servers are needed — but they should be equipped with dedicated graphics hardware since even consumer-grade graphics cards are by a factor 10–20 faster than pure software rendering. Finally, the client needs a display, and this can be a tiled display or even a complex CAVE setup (with the Paraview GUI running on a remote monitor).

However, these subtasks can also be arbitrarily combined — the stand-alone Paraview application (*paraview*) is a combination of these three components, the server (*pvserver*) can act both as a data server and a render server, and the client (*pvclient*) can also do the rendering (by deactivating the switch “Composite” in the menu item “3D View Properties”). Furthermore, each of these three components can also be used in parallel with MPI.

As an example, let’s consider a large dataset that is made up of 32 independent parts. Because of memory issues, it cannot be loaded on a single machine. The client is an ordinary PC, and a computing cluster acts as a data server:

Listing 5.2: Example: client-server operation on a cluster and a workstation

```
user@clusterhead$ mpirun -machinefile machinefile.txt -np 32 paraview64/bin/
    pvserver --use-offscreen-rendering

user@workstation$ paraview32/bin/pvclient -sh=clusterhead &
```

In this case, even two different Paraview versions (made from the same source code version!) can be used. The cluster nodes use 64-bit processors and have no graphics card, so a 64-bit version of Paraview with MPI and Mesa library is used. In contrary, the workstation has a 32-bit processor and a 3D-accelerated graphics card, so it uses an appropriate pvclient version which does also the rendering. Since it is not MPI-enabled, the level-of-detail feature is available which allows smooth handling with reduced resolution. If necessary, one could also switch to server-side rendering by activating the switch “Composite”. In this case, it can be reasonable to use separated render servers because the rendering process consumes some memory which would not be available for the data.

If a cluster is used, there is an additional difficulty. Because the cluster nodes usually form an isolated subnet which can only be reached from outside via the cluster head, there is only one network connection possible, from the data server (represented by the cluster head as the first data server machine) to the single client. Server-side rendering on the cluster as well as multiple rendering clients or render servers outside the cluster is not possible, because multiple network connections between the render servers/client(s) outside and the data or also render servers inside the cluster would be necessary. So only a single client is possible (either outside the cluster or on the clusterhead with X forwarding to a machine outside) which does the rendering and hence is the bottleneck of this configuration; in the last resort the machine which finally displays the data has to be included into the cluster network.

5.4. Offscreen Rendering

Remote rendering processes open X windows to draw their images. If a process cannot open an X window, it refuses to run. The remedy for this problem is offscreen rendering (see section 4.3), which is activated with the option `--use-offscreen-rendering`. Especially if client-side rendering is used because no network connection between the render servers and the client can be established (see above), this option needs to be activated because otherwise the server processes cannot even get started (even if no server-side rendering is actually used).

5.5. Simplification of the Data

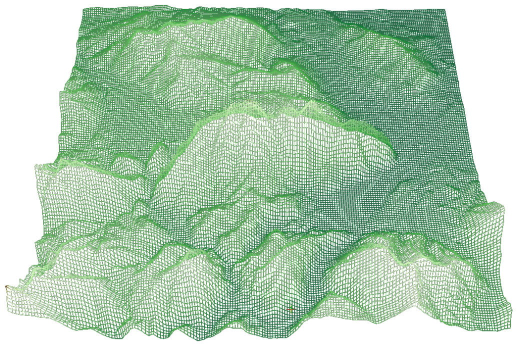
The full amount of data is not always needed. Because only the outer surface is visible, it is sufficient to save only the surface; if it is not necessary to edit the dataset afterwards, the rest can be omitted. By skipping one dimension, complex datasets which normally require a cluster to be loaded, can be shown with full resolution (at the surface) on a single machine, and animations run fluently.

But also surfaces can be simplified. The following example (figures 5.1 and 5.2) depicts two different ways to do this: In the first image, every other gridpoint has been omitted, which reduces the data to a fourth. In the second image, Paraview's "Triangulate" filter has been applied, which doubles the number of polygons by cutting them into triangles; afterwards, the "Decimate" filter has been applied to reduce the polygons to 12 % of their original number, which results in approximately the same memory size as the first image (and even a slightly smaller file size) — but more visible details, since the adaptive algorithm spends more points at regions with high curvature. Furthermore, using the "Triangle Strips" filter, the number of cells (and accordingly the memory consumption and filesize) can be reduced by half.

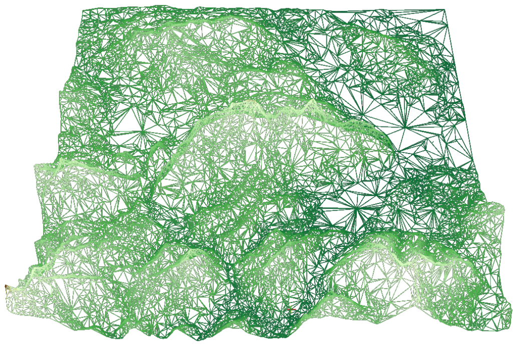
5.6. Tools for Parallel Processing

If there is no dedicated visualization cluster available, other PCs in the network have to be used. Because they are used by many people for different tasks, it is important to know which machines have enough free resources. The following script queries all computers listed in a given

Figure 5.1: Different ways of data reduction at the surface grid of Mt. Hochstaufen



(a) Every other grid point has been omitted



(b) Adaptive triangular grid, 88 % of the original triangles have been omitted

file for their resources (CPU speed, CPU load, total memory and memory usage) and displays them in a table, ordered by CPU load and memory usage. A machinefile can be generated automatically from the first machines listed in the table.

Listing 5.3: *getmachinestatus.pl*

```
#!/usr/bin/perl

my $procs = 0;          # current number of parallel processes
my $hostname = "";      # the current hostname
my $hostnames_file = 'hostnames.txt';      # file with all available hostnames
my @childpipes;         # pipelines to all child processes

open( HOSTNAMES, "< $hostnames_file" ) || die "Error: cannot open hostnames file
    $hostnames_file";
open( SORTPIPE, "| sort -k5,5n -k4,4nr -k3,3nr" ) || die "Error: cannot open
    pipeline";

printf( STDERR "%-15s\t%8s\t%7s\t%7s\t%7s\t%-15s\n", "Hostname", "CPU(MHz)", "mem(
    MB)", "memfree", "CPUload", "users" );

# with every line = every hostname
while( $hostname = <HOSTNAMES> )
{
        chomp( $hostname );

        # limit number of processes
        #if( $procs == $maxprocs )
        #{
        #       wait();
        #       $procs--;
        #}

        # fork process; child process has PID 0
        #$pid = fork();
```

```perl
        $pid = open( $childpipes[ $procs ], "-|" );     # fork a child process and create a reading pipeline to it

        # child process
        if( 0 == $pid )
        {
                $processor = `ssh -x $hostname 'grep "cpu MHz" /proc/cpuinfo' | uniq`;
                $processor =~ s/.*: *([.[:digit:]]+).*/\1/;
                $processor += 0;

                $totalmemory = `ssh -x $hostname 'grep "MemTotal" /proc/meminfo'`;
                $totalmemory =~ s/.*: *([.[:digit:]]+).*/\1/;
                $totalmemory /= 1024;   # MB instead of kB

                $status = `ssh -x $hostname 'vmstat -S M 5 2'`;
                $status = (split( /\n/, $status ))[3];
                $freemem = (split( /[[:space:]]+/, $status ))[4] + 0;
                $cpuavg = (split( /[[:space:]]+/, $status ))[13] + 0;

                $users = `ssh -x $hostname 'w -h' | cut -d' ' -f1 | sort | uniq | grep -v $ENV{USER}`;
                $users =~ s/\n/ /g;

                printf( "%-15s\t%6d\t%6d\t%3d\t%3d\t%-15s\n", $hostname, $processor, $totalmemory, ($freemem / $totalmemory * 100), $cpuavg, $users );
        }
        # parent process
        elsif( -1 != $pid )
        {
                $procs++;
        }
}

if( 0 != $pid ) # only for the parent process
{
        for( $i = 0; $i < $procs; $i++ )
        {
                $pipe = $childpipes[ $i ];

                # receive the messages from all child processes and feed them into the sort pipeline
                while( <$pipe> )
                {
                        printf( SORTPIPE $_ );
                }
        }

        close( SORTPIPE );

        # wait until all processes have terminated
        do
        {
                wait();
        } until( -1 == $? )
}
```

Not only simulations and visualizations, but also some intermediate processing steps can benefit from parallelization. For example, writing ASCII VTK files is easier to program than XML files, especially if the data is stored in binary, compressed with the zlib and base64-encoded. But only files in the XML format can be bundled as one parallel file. With, say, 128 data files, it takes some time on a single PC to convert them into the XML VTU format — hence it makes sense to distribute this task on several machines. This can be done with MPICH2; *mpiexec* starts also non-MPI programs like shell scripts, which do not need interprocess communication. The total number of processes and the MPI rank can be determined from the environment variables `PMI_SIZE` and `PMI_RANK` or `MPD_JSIZE` and `MPD_JRANK`. However, MPICH2 is not necessary; it can also be done using the following shell script:

Listing 5.4: *execute-parallel.sh*

```
#!/bin/sh

# some initializations
MACHINEFILE="machinefile.txt"
NUM_JOBS=128
COMMAND=$PWD/'pargrid+data2vtu.sh;'     # use always absolute paths; command line
    must end with a semicolon

# internal variables
JOBS_PER_MACHINE=$(($NUM_JOBS / $NUM_MACHINES + 1))
NUM_MACHINES=`wc -l < $MACHINEFILE`

echo "Starting the jobs ..." > /dev/stderr

# start the jobs via SSH
for i in `seq 1 $NUM_MACHINES`; do

        # clear command list
        COMMANDLIST=""

        # calculate first and last job number on this machine
        FIRSTJOB=$i
        LASTJOB=$(($FIRSTJOB + $JOBS_PER_MACHINE * $NUM_MACHINES))
        if [ $LASTJOB -gt $NUM_JOBS ] ; then LASTJOB=$NUM_JOBS ; fi;

        # assemble the command list by repeating the command with the
            corresponding job numbers
        for j in `seq $FIRSTJOB $NUM_MACHINES $LASTJOB`; do
                COMMANDLIST="$COMMANDLIST export job=$j; $COMMAND"
        done

        # extract the current machine name from the machinefile and send the
            command list to this computer
        MACHINE=`sed -n -e "s/:.*//; ${i}p" < $MACHINEFILE`
        ssh -x $MACHINE "export machine=$i; export workingdir=$PWD; $COMMANDLIST"
            &
done

echo "All jobs are started." > /dev/stderr
```

5.7. Putting It All Together: How to Process a Dataset

In this example, a dataset from a mantle convection simulation is processed.

First, the grid — if not available — has to be generated. This is done with a modified version of TERRA's *parplt.f* program:

```
/import/two-data/parplt-Code$ vi size.h
/import/two-data/parplt-Code$ make clean
/import/two-data/parplt-Code$ make parplt-parallel
/import/two-data/parplt-Code$ mpirun -np 128 -machinefile machines.LINUX ./parplt-
    parallel.x
/import/two-data/parplt-Code$ for i in pargrid???; do gzip -c $i > ../GRIDS
    /128-16/$i.gz; done
```

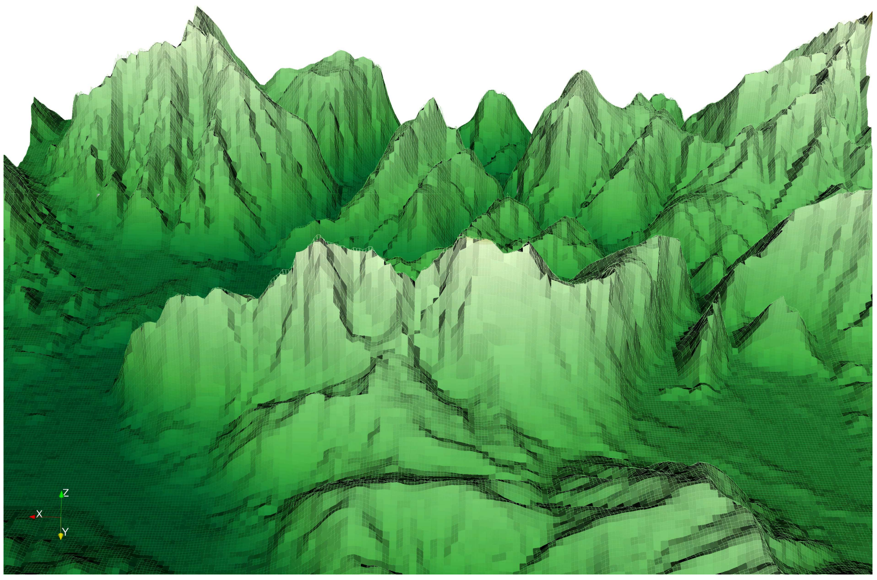
Next, the correct filenames and paths have to be entered into the conversion scripts; it is important to use absolute paths:

```
/import/two-data/MPI-Tools$ vi pargrid+data2vtu.sh
/import/two-data/MPI-Tools$ vi execute-parallel.sh
/import/two-data/MPI-Tools$ ./execute-parallel.sh
```

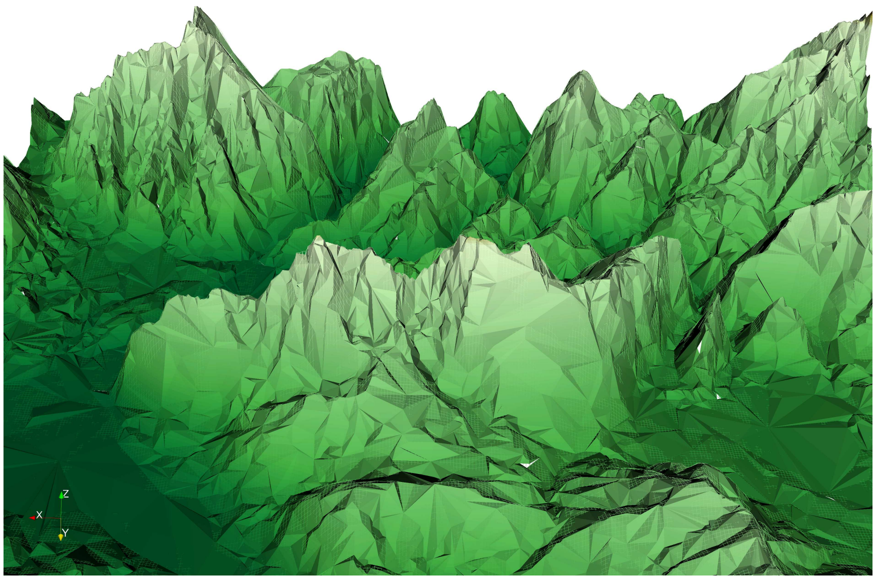
Finally, the pvtu file has to be created:

```
/import/two-data/MPI-Tools$ vi create_pvtu.sh
/import/two-data/MPI-Tools$ ./create_pvtu.sh
```

Figure 5.2: Different ways of data reduction; both images have been overlaid with the full-resolution wireframe model. The first model resembles the original model but looks quite blurred, whereas the second model has a much higher resolution at the edges



(a) Every other grid point has been omitted



(b) Adaptive triangular grid, 88 % of the original triangles have been omitted

6. Examples

What I cannot create, I do not understand.
Richard Feynman

6.1. Simulation of Mantle Convection

6.1.1. Introduction

TERRA is the code for the simulation of mantle convection that is used at the Geodynamics group at the LMU. It is a parallelized finite element code for solving the Navier-Stokes equation using the multigrid approach [Bunge and Baumgardner, 1995]. The simulation is done on a PC cluster consisting of 128 AMD Opteron processors; a typical simulation setup spans over 84 million grid points which yields a spatial resolution of less than 75 km.

The spherical grid layers are approximated by icosahedra (made up of spherical triangles; see figure 6.1); every two adjoining triangles form one diamond which is recursively quartered into subdomains [Baumgardner and Fredrickson, 1985], so every process of the running program owns one subdomain on every diamond or every other diamond (depending on the number of subdomains, which must be a power of four, whereas the number of processors must be a power of two). Every processor writes its output into an ASCII file, which results in approximately 4 GB of data per snapshot in the setup described above (64 subdomains per diamond with 32^2 grid points each, 128 radial layers).

A second program (*parplt.f*) that uses the same configuration files as the actual TERRA simulation code computes a grid, reads the simulation files in parallel, concatenates them and outputs them together with some header information and the grid data as a VTK unstructured grid file.

6.1.2. Taming the Data

Two sorts of output files are generated: High-resolution data files and downsampled files. The latter are generated by omitting every other data point in each direction, thus giving roughly 10 million grid points or about 700 MB raw ASCII data. Because also the grid points have to be written (three coordinates per point), the cell connectivity (six point indices per cell, 20 million cells) and some other data, the ASCII file size exceeded 2 GB which seemed to be a problem for the Fortran code that generates the VTK file (presumably a size limit for common blocks); the output was truncated. With some tweaking in the output routines, the file size could be

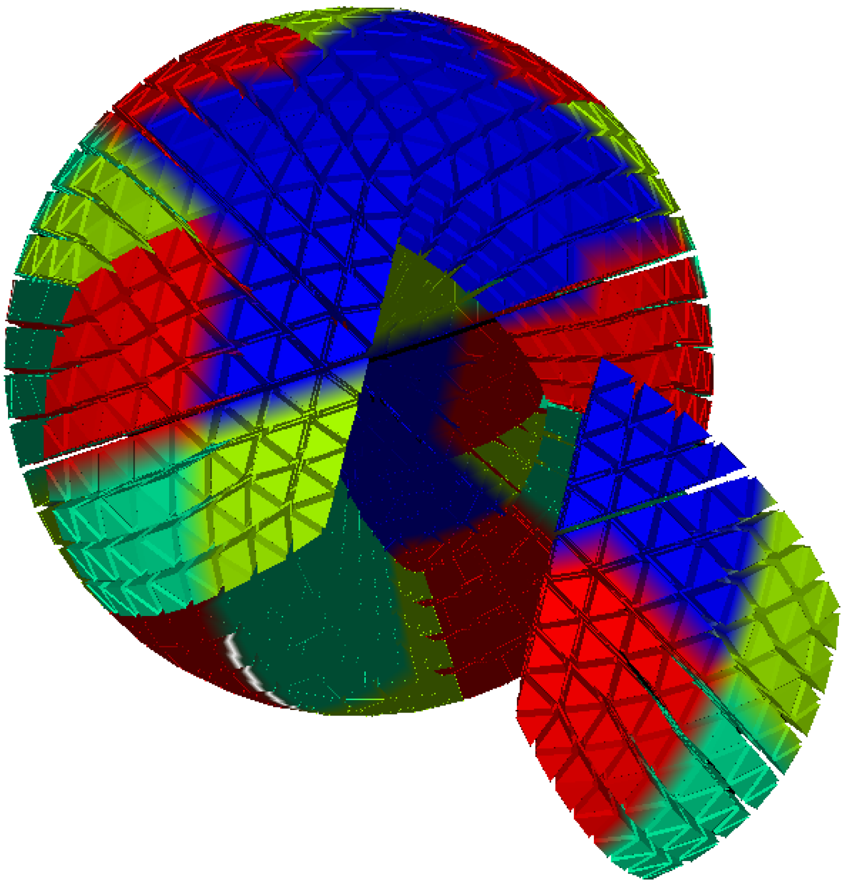


Figure 6.1: Icosahedral grid used by the TERRA code; the different colors denote the subdomains used for the parallelization.

reduced to just below 2 GB, but it took still about 30 minutes to load the downsampled data file — without being able to display it (on a single PC).

The next step was to switch the code to binary output. This required the *BINARY* file mode which is proprietary to Intel Fortran compilers; portable code only permits the *UNFORMATTED* (i.e. ASCII) and *FORMATTED* (i.e. binary, with a fixed record length) file type. Using this Intel extension, the file size could be cut down to 800 MB, whereas also some cleaning in the code was required — binary files do not forgive one single redundant byte. Loading the data file with Paraview became also much faster, it took only a few minutes.

Listing 6.1: Example of binary and ASCII output with Fortran

```
      if (BINARY_OUT == 1) then ! CM
        open (nf, file='pargrid',form='binary',status='unknown', ! CM
     &   convert='BIG_ENDIAN') ! CM
      else ! CM
```

```
          open (nf, file='pargrid',form='formatted',status='unknown')
        end if ! CM
```

```
        if (BINARY_OUT == 1) then ! CM
          write(mystring,'(a5,i12,i12)') ! CM
       &   'CELLS',icellnum,icellnum*(ipointspercell+1) ! CM
          write(nf) trim(mystring), char(10) ! CM
        else ! CM
          write(nf,*)'CELLS',icellnum,icellnum*(ipointspercell+1)
        end if ! CM
```

Although the file size was now low enough and the file could be loaded much faster, it could still not be displayed; at least, the newest Paraview version at that time (2.4.2) did not crash instantaneously, but displayed a box outlining the dataset instead of its surface. Obviously there was just enough memory to load the raw data, but not to process or display it.

The next attempt was to use the parallel capabilities of Paraview. For that, a MPI-enabled version had to be compiled and to get started (see section 4.2). After that, the VTK file could be loaded, when running Paraview on about 10 machines. (Sometimes, this did not work either; the error messages are not very instructive, but the experience seems to indicate memory problems. Although the PCs have more than enough memory to accomodate the raw file, the rendering pipeline seems to consume a multiple of this amount.) Obviously, every involved PC needs enough memory to store the whole data file; the rendering is distributed over all machines, so the additional memory requirement per machine remains small.

After loading the file, the so-called "D3" filter ("distributed data decomposition") can be called which splits up the data file, so every participating computer holds one spatially separated part of the grid. Then, the data can be saved as one of VTK's parallel file types, made up of single files with the subsets from the MPI nodes. Now, the original file can be unloaded; since the parts of the newly created PVTU file are loaded in parallel from the machines in the MPI cluster, memory is not a problem anymore — every machine holds now only a small part of the original file.

Although the visualization of the downsampled grid with 10 million points is now possible, the bottlenecks are obvious: The file has to fit into the memory of every single computer in the MPI cluster, and writing files bigger than 2 GB also seemed to be a problem (whereas this should be solvable; neither the hardware nor the OS have this restriction). Also, the handling was not very fast. The solution is obvious: Instead of collecting the data into one big file which brings the PCs to their limits, the grid and the data should be written into distributed files, linked together by a PVTU master file. The data is already available in separate files for each processor; the output program *parplt.f* had to be modified such that every process creates its own subgrid and writes it into a separate file instead of writing one monolithic grid file. Since the original data files are ASCII files, the grid has also been written in ASCII. Then, a simple shell script concatenates the subgrids with their corresponding data sections and calles the following Tcl script that converts the ASCII VTK files to binary VTU files — which can be enclosed by a PVTU envelope. With Paraview running on multiple machines, this PVTU file can be loaded at once, or if the cluster is not available, the constituent parts can be viewed from a single PC. With this method, it was even possible to load the full resolution dataset into Paraview.

6.1.3. Temperature Distribution in the Earth's Mantle

This simulation uses the TERRA code described above. The setup includes internal heating (no heating from the CMB), viscosity increasing with depth [Bunge *et al.*, 1997], and as a boundary condition a surface temperature of 300 K.

The data has been loaded with a parallel running 64-bit MPI version of *pvserver* and displayed with a non-MPI version of *pvclient* on a 32-bit machine. With one spherical clipping operation, the uppermost layer (1 % of the radius) has been removed since the thermal boundary condition defines equal surface temperature, so the convection structures form below. A second clipping operation has removed one hemisphere, thus showing a cross-section (see figure 6.2). After applying the filter "Extract Surface", this image can be saved as a polygonal surface and then also be loaded from a single PC.

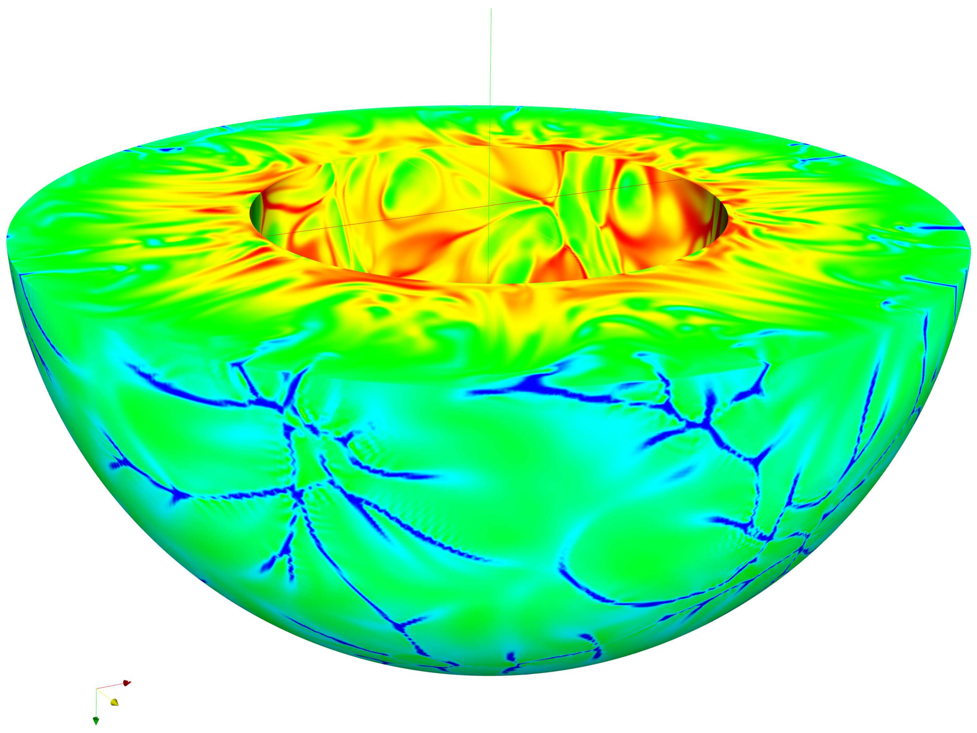


Figure 6.2: This image shows the temperature in a convection simulation; the viscosity increases with depth, hence the flow forms branches near the surface before sinking down [Bunge *et al.*, 1996]. The outer layer (1 % of the radius) is removed because of the boundary condition. There is only internal heating, thus there are no plumes originating at the CMB, whereas the surface is cooled externally, so there are plume-like structures dipping down.

6.1.4. Seismic Velocities in the Earth's Mantle

Since the initial conditions for the earth's mantle convection are not known, forward-modelling can only provide the general development of plate tectonics, but not the current state. For the simulation of the recent convection, the inverse problem can be solved; giving the current state and the recent plate motions as boundary conditions, a simulation using the data assimilation method computes the properties of a convection that leads to the current state. This has been done with the program TERRA (mentioned above); see [Bunge *et al.*, 2002].

The output file contains the seismic velocities in the mantle, with normalized mean velocities in each radial layer. With only 1.3 million grid points, this file could be loaded on a single PC. Two isosurfaces at velocity variations of $-3\,\%$ resp. $+2\,\%$ then outline the subducting slabs (= cold material with increased seismic velocity) and the mid-ocean ridges (= warm material, low velocity zone) (see figure 6.3). Using the "Smooth" filter, the surfaces can be smoothed, and saved as a polygonal file. Together with an overlaid topography file or with plate boundaries, this gives quite impressive models of what structures can be expected underneath plate boundaries and orogenes.

6.2. Seismic Wave Propagation

6.2.1. Introduction

The wave propagation code for this simulation uses a staggered-grid finite-difference scheme on the elastic wave equations in a spherical section of the earth [Igel *et al.*, 2002]. Andreas Fichtner has parallelized this code and added support for anisotropy and attenuation. Two test runs have been done, one with anisotropy only in z-direction, and one with a PREM model. The volume is partitioned into 15 (PREM) resp. 18 pieces, each computed by one processor.

6.2.2. Visualization

First, the coordinate grids had to be created. The steps along the coordinate axes, given as polar coordinates, were combined to triplets for every grid point and converted into cartesian coordinates; this has been done with a shell script (similar to listing 6.2). Then, the corresponding data was appended to every created VTK grid file and then converted into the VTU format via Tcl script (see listing 3.2). Finally, a PVTU file links all the sections into a big file (see figure 6.4).

6.3. Topography

The geosciences deal not only with abstract and self-contained simulations, but have often to display features in the context of their surrounding. Maps are often essential parts of

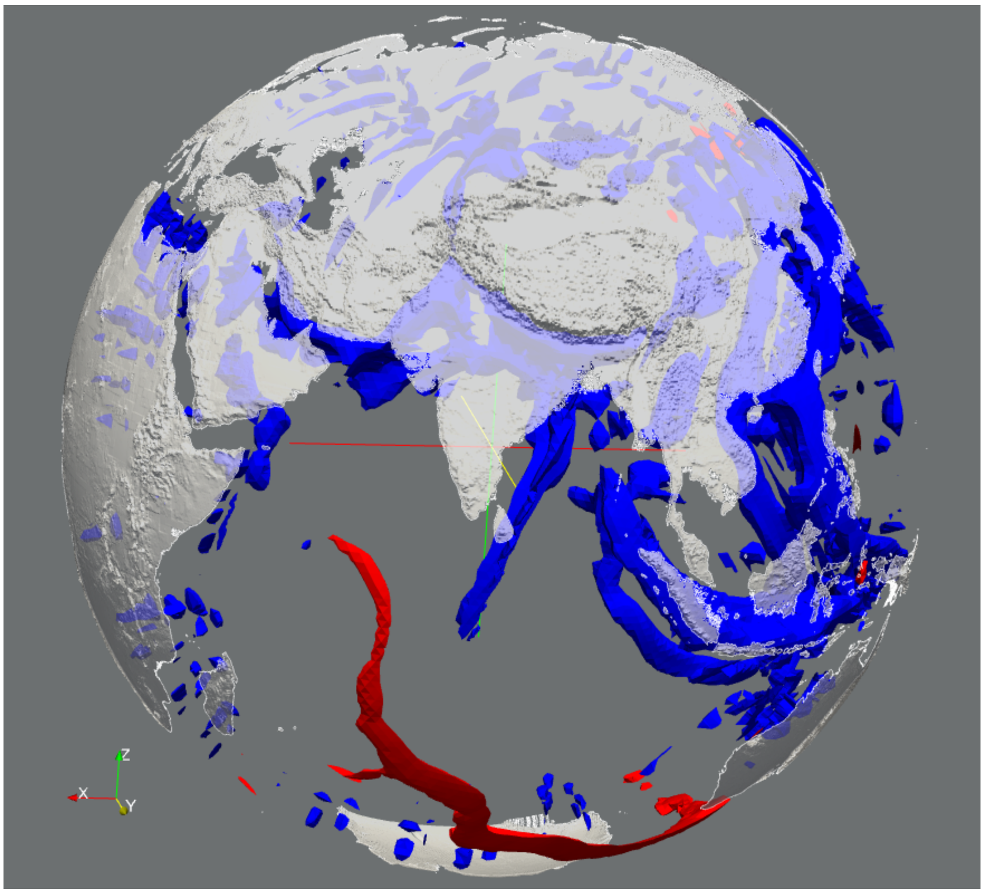


Figure 6.3: Seismic velocities in the earth's mantle (computed from the simulated temperatures); red: middle ocean ridges, where the seismic velocity is reduced more than 3 %; blue: subducting slabs, where the seismic velocity is increased more than 2 %.

visualizations; in these cases, external map data has to be converted to 3D structures, more precisely: polygonal surfaces.

6.3.1. The Globe, Created From the ETOPO2 Dataset

The ETOPO2 dataset is a global elevation grid, compiled from a variety of sources, with a resolution up to 2 arc minutes. It is available in various formats, including GMT grid file and ASCII table.

In this example, the basis was a GMT grid file with a resolution of 5 arc minutes. To convert it to a VTK file (see figure 6.5), the following steps were necessary:

- With the tool *grd2xyz* the grid file can be converted to an ASCII table.

- Because the GMT version did not contain the poles, these data points had to be added by hand — i.e. for latitudes of ±90° the coordinates of ±89° were copied and the heights were changed to those of the north pole or south pole, respectively.
- Converting the grid to cartesian coordinates: This is done by a simple AWK script which takes the longitude and latitude values and converts them to points lying on the unit sphere. Additionally, the elevation values can be added to the earth radius to create a spatial relief on the sphere; an exaggeration factor of 25 gives quite impressive (yet not realistic) results.
- Create a VTK file: In the next step, the VTK header is printed, then the grid points are appended. After that, the connectivity has to be created. The polygons are trapezoids, made up of two adjacent points and the neighboring two points on the next latitude step. Then the height data (i.e. the third column of the `*.xyz` file) is added. Because custom color tables only seem to be possible for values between 0 and 1, the heights have to be normalized to this range.
- Finally, the color table with 256 entries is added.

A shell script was created for the conversion:

Listing 6.2: Converting the ASCII table to VTK

```
#!/bin/sh

INPUTFILE=$1    # input: first filename given on commandline
RADIUS=6371000  # radius of earth in m
EXAGGERATE=25   # set to 0 for no topographical exaggeration; max. value ca. 50
SHIFT_Z=133     # shift altitude scalars by this amount (in m), so the color table
     matches
#SHIFT_Z=0      # shift altitude scalars by this amount (in m), so the color table
     matches
IS_SPHERE=1     # determines if the map is a sphere, i.e. that lon=360Â° should be
     connected with lon=0Â°
POLAR2CART=1    # convert polar coordinates to cartesian coordinates (use the
    given earth radius)

LONSTEPS=`cut -f1 < $INPUTFILE | sort | uniq | wc -l`
LATSTEPS=`cut -f2 < $INPUTFILE | sort | uniq | wc -l`
NRPOINTS=`wc -l < $INPUTFILE`

# output vtk header
echo "# vtk DataFile Version 3.0"
echo "xyz2vtkpoly.sh"
echo "ASCII"
echo "DATASET POLYDATA"
echo "POINTS $NRPOINTS float"

# output grid points

if [ 1 -eq $POLAR2CART ]; then
# convert to cartesian coordinates
        awk "BEGIN {\
                PI = 3.14159265359;\
        }\
        {\
                R = (($RADIUS + \$3 * $EXAGGERATE) / $RADIUS);\
                \
                X = R * cos(\$1 / 180 * PI) * cos(\$2 / 180 * PI);\
                Y = R * sin(\$1 / 180 * PI) * cos(\$2 / 180 * PI);\
                Z = (R * sin(\$2 / 180 * PI));\
                \
                \$1 = X;\
                \$2 = Y;\
                \$3 = Z;\
                \
                print;\
        }" < $INPUTFILE
else
        awk "{ \$3 *= $EXAGGERATE; print; }" < $INPUTFILE
fi

# output polygons
if [ 1 -eq $IS_SPHERE ]; then
        echo "POLYGONS $(($NRPOINTS - $LONSTEPS)) $((5 * ($NRPOINTS - $LONSTEPS)))"
else
        echo "POLYGONS $(($NRPOINTS - $LONSTEPS - $LATSTEPS + 1)) $((5 * ($NRPOINTS - $LONSTEPS - $LATSTEPS + 1)))"
fi


#awk "BEGIN { for( i = 1; i < $NRPOINTS - $LONSTEPS; i++ ) printf( "'"'"4\t%d\t%d\t%d\t%d\n"'"'", i, (i+$LONSTEPS), (i+$LONSTEPS+1), (i+1) ); }"
awk 'BEGIN {\
for( lat = 0; lat < '$LATSTEPS' - 1; lat++ )\
{\
        for( lon = 0; lon < '$LONSTEPS' - 1; lon++ )\
        {\
                printf( "4\t%d\t%d\t%d\t%d\n", (lat * '$LONSTEPS' + lon),( (lat + 1) * '$LONSTEPS' + lon), ( (lat + 1) * '$LONSTEPS' + lon + 1), (lat * '$LONSTEPS' + lon + 1) );\
        }\
        \
        if( 1 == '$IS_SPHERE' )\
        {\
                printf( "4\t%d\t%d\t%d\t%d\n", (lat * '$LONSTEPS' + lon),( (lat + 1) * '$LONSTEPS' + lon), ( (lat + 1) * '$LONSTEPS'), (lat * '$LONSTEPS') );\
        }\
}\
}'

# output height data
echo "POINT_DATA $NRPOINTS"
echo "SCALARS altitude float"
echo "LOOKUP_TABLE topo"
awk "{ \$3 += 9652 + $SHIFT_Z; \$3 /= (9652 + 6401); print \$3; }" < $INPUTFILE
#awk "{ \$3 -= 423; \$3 /= ((1860 - 423) / 0.390456612471); \$3 += 0.609543387529; print \$3; }" < $INPUTFILE # Hochstaufen: min 423.6, max 1859.2
#awk "{ print \$3; }" < $INPUTFILE

```

```
# output lookup table
cat <<EOF
LOOKUP_TABLE topo 256
```

6.4. Hochstaufen

6.4.1. Introduction

Hochstaufen is a mountain in Bavaria near Bad Reichenhall where earthquake swarms occur. Thus, a dense network of seismometer stations has been installed around the mountain, and several swarms could be recorded in the recent years. Fluids have been suspected to play a role in the emergence of earthquake swarms for a long time, yet the proof has been difficult. Contrary to other swarm quake areas, where volcanic activity is assumed to drive the fluids into the rock, there is no volcanism around Hochstaufen — the fluids come from above, making this mountain an ideal study area by installing seismometers and meteorological stations. In 2005, it could be shown for the first time that there is a strong correlation between heavy precipitation and earthquake activity at Hochstaufen [Kraft *et al.*, 2006b]; the data comes mostly from the year 2002 where there has been torrential rain in March and August, each followed by an earthquake swarm.

6.4.2. Visualization

1171 events have been recorded in that year, for 612 of them a hypocenter location could be determined [Kraft *et al.*, 2006a]. Since the exact mechanisms are still unknown, the spatial and temporal distribution of the hypocenters can be informative. The basis for an 3D animation of the swarm quakes is a list of all earthquakes (with coordinates given as latitude/longitude and UTM, depth, date, day of the year and magnitude) and a topographic map given in the GMT grid file format. The grid file has been converted with the GMT tool *grd2xyz* to an ASCII table and then processed like described above. Since the coordinates were given in the UTM coordinate system, no conversion to cartesian coordinates was necessary. The result is a VTK file with a rectangular grid (see figure 5.1).

For the earthquakes, a two-stage process was necessary. First, the coordinates and magnitudes of the earthquakes were written into a VTK file; any information about the point connectivity has been omitted. This file has then been fed to the following Tcl script which calls VTK functions to generate spheres at all gridpoints (see figure 6.6):

Listing 6.3: Creating glyphs at every grid point (the input and output filenames are given as command line parameters)

```
#!/usr/bin/env vtk

# include VTK
package require vtk

# read a VTK polydata file
```

```
vtkPolyDataReader reader
        reader SetFileName [lindex $argv 0]

# create a sphere
vtkSphereSource sphere

vtkTransform transform
        #transform Translate 0.5 0.0 0.0

# convert the sphere to polydata
vtkTransformPolyDataFilter transformfilter
        transformfilter SetInput [sphere GetOutput]
        transformfilter SetTransform transform

# create glyphs on every gridpoint; use the spheres as glyphs
vtkGlyph3D glyph
        glyph SetInput [reader GetOutput]
        glyph SetSource [transformfilter GetOutput]
        glyph SetVectorModeToUseVector
        glyph SetColorModeToColorByScalar
        glyph SetScaleModeToScaleByVector
        glyph SetScaleFactor 300

# write a binary VTK polydata file
vtkPolyDataWriter writer
        writer SetInput [glyph GetOutput]
        writer SetFileName [lindex $argv 1]
        writer SetFileTypeToBinary
        writer Write

exit
```

Not only the spatial distribution, also the temporal relocation of the hypocenters is of interest. This can be shown with an animation. The described procedure was repeated for every frame of the animation, in total 1000 frames. In every frame, only those quakes were included where the day-of-the-year value was smaller than $i/1000 \cdot 365$ (with i as the loop counter).

6.5. Magnetic Field Reversal

This example showing a magnetic field reversal has been provided by Roman Leonhardt. Starting from paleomagnetic data, he has modeled the magnetic field on the core mantle boundary and on the earth's surface and cast the results in VTK files over 1000 timesteps.

6.5.1. Changing the Color Table

The main problem was Paraview's default rainbow color scale. This gives the best optical distinction between the data values, yet it is not appropriate for the magnetic field strength since it suggests a continuous range of values without any symmetry. The two polarities should rather be expressed by two different colors (usually red for normal and blue for inverse polarity), where the saturation indicates the field strength. To achive this, the data values have to be normalized to the range [0 . . . 1] and the new color table has to be appended — in every file. This has been done with the following awk script which simply uses a square root function to generate a nonlinear color table:

Listing 6.4: Scaling the data values; the script looks for the beginning of the data section and scales every value according to the maximum and minimum values that have been passed to the script; the conversion stops at the next blank line.

```
#!/usr/bin/awk -f

BEGIN {
        min = ENVIRON["min"];
        max = ENVIRON["max"];

        #RS = backup_RS;
        RS = "\r\n";    # because original file has windows line breaks

        do_scale = 0;
}

# skip lines until "LOOKUP_TABLE" appears
/^LOOKUP_TABLE/ {
        do_scale = 1;
        print "LOOKUP_TABLE france";
        getline;
}

# after the next empty line: stop again
/^$/ {
        do_scale = 0;
        getline;
}

# scale the values
{
        if(1 == do_scale)
        {
                $1 = ( ($1 - min) / (max - min) );
        }

        # remove windows line breaks; but faster is: tr -d '\r'
        #$0 = $0;

        print;
}
```

The new color table is computed the following way:

Listing 6.5: Creating a nonlinear color scale, ranging from blue over white to red

```
#!/usr/bin/awk -f

BEGIN {
        # see also: makecpt -Cpolar -N -Z -T0/1/0.01 | awk '(NR > 3) { NF = 4; $1
            = $2 / 255; $2 = $3 / 255; $3 = $4 / 255; $4 = 1; print }'
        steps = 100;

        print "LOOKUP_TABLE france " (2 * steps);

        for( i = steps; i > 0; i-- )
        {
                fraction = sqrt( i / steps );

                $1 = (1 - fraction);
                $2 = (1 - fraction);
                $3 = 1;
                $4 = 1;

                print;
        }

        for( i = 0; i < steps; i++ )
        {
                fraction = sqrt( i / steps );

                $1 = 1;
                $2 = (1 - fraction);
                $3 = (1 - fraction);
                $4 = 1;

                print;
        }
}
```

6.5.2. Creating Colored Glyphs

Creating vector arrows from the vectorial surface data is easy; but in this step, the color table gets lost. So it is not sufficient to create the arrows at runtime via the Paraview glyph filter, but it has to be done non-interactively. With a Tcl script similar to listing 6.3 VTK files with vector arrows can be generated; afterwards, the color table is added the same way as described above. For the initial and final image see figure 6.7.

6.5.3. Saving Complex Sceneries

This example scenery is made of several files:

- the magnetic field strength at the core mantle boundary (*radcomp_binary_*.vtk*)
- the magnetic field vectors at the surface (*survec_glyph_binary_*.vtk*)
- the continents, semi-transparent (*continents_45min_threshold_061.vtk*; derived from the topographic map with a threshold filter of ≥ 0.61)
- the continent boundary lines (*continent_boundaries_45min_cut_sphere_1830.vtk*; derived from the topographic map, cut with a sphere with radius 1.83)

- longitude and latitude lines on the CMB (*long_lat_cmb_edges.vtk*)

Paraview does not provide any means to save "project files" (containing the files to load), but "state files" which are actually Tcl scripts which memorize every file, filter or parameter the user has loaded or applied. However, filenames are stored with their complete pathname, so these files are not portable. Changing the absolute filenames to relative pathnames did not work; but dynamically adding the current absolute path is a solution. This can be done by replacing the full path, e.g.

```
"/home/user/subdirectory/file.vtk"
```

by

```
[join [list $env(PWD)"/file.vtk"] ""]
```

.

6.6. Earthquakes in Subduction Zones

In this example, the isosurface from section 6.1.4 was re-used, which outlines very well the subducting slab under the Andes. It was combined with the world map from section 6.3.1, cut out and combined with beach balls indicating the hypocenter locations and fault planes (see figure 6.8).

6.6.1. Creating Beach Balls

A beach ball is a sphere with a black-white color table. Its cartesian coordinates are calculated from polar coordinates in two nested loops, with a stepwidth of 10° in latitude and longitude. For longitudes between 90° and 180° and between 270° and 360° the color of the surface point is set to white, otherwise to black. The polygon connectivity is written in two further nested loops; every two adjacent points are connected with the two points on the next parallel to a trapezoidal polygon.

6.6.2. Orientating Beach Balls at Hypocenters

The hypocenter locations are taken from the Harvard CMT search page [Harvard CMT, 2006]; using the script *ndk2table.awk*, the file is converted to a table with the location, magnitude, fault plane orientation and time of each earthquake on one line. For this example, the output has been constrained to magnitudes above 5.0, latitudes from the equator to −60° and longitudes between −60° and −90°.

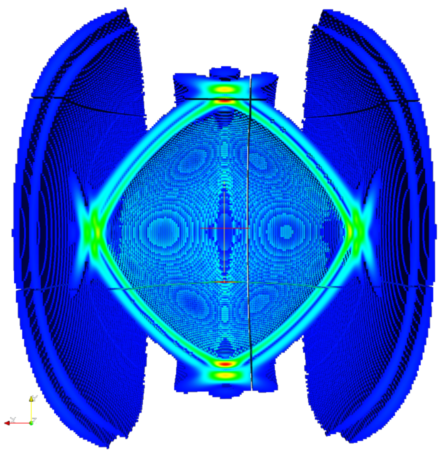
The beach balls are created in the origin. Then the rotations around the fault plane coordinates are carried out, using rotary matrices; first comes a rotation about the rake angle around the x axis, then about the dip angle around the z axis, and finally about the strike angle, again around the x axis.

After that, the beach ball is rotated about its latitude value around the y axis and about its longitude value around the z axis and translated to the earthquake location by adding the corresponding coordinates.

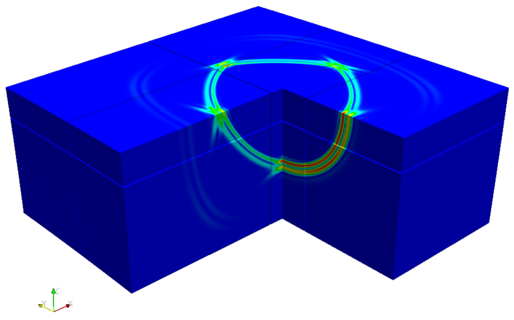
The following script performs the described steps to create a beach ball for every earthquake in the previously generated table.

Listing 6.6: *make_beachballs_from_file.awk*; input = filename given on the command line, output = stdout

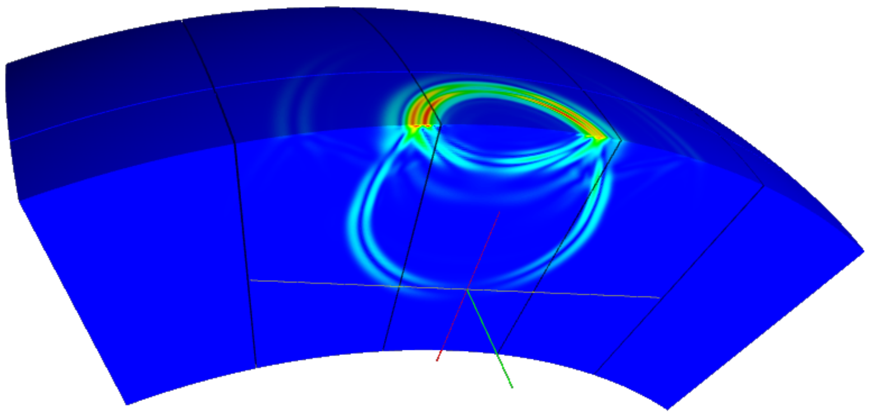
```
#!/usr/bin/awk -f

BEGIN {

# initializations
NUMBER = 0;              # number of beach balls
STEPWIDTH = 10;          # must be a divisor of 360 and 190 => 1, 2, 5, 10
BB_RADIUS = 0.01;        # compare to earth radius
EARTH_RADIUS = 1;

# local variables and settings
#FS  = ",";     # input: separated by komma
OFS = "\t";     # output: separate fields with a tab
PI = atan2( 0, -1 );
LONSTEPS = int( 360 / STEPWIDTH );
LATSTEPS = int( 190 / STEPWIDTH );
NRPOINTS = LONSTEPS * LATSTEPS;          # number of grid points per beach ball
CORR = -90;

FILENAME = ARGV[1];      # use input from first command line argument
("wc -l < " FILENAME) | getline NUMBER; # get number of lines in input file

# output vtk header
print "# vtk DataFile Version 3.0";
print "make_beachballs_from_file.awk";
print "ASCII";
print "DATASET POLYDATA";
print "POINTS " (NRPOINTS * NUMBER) " float";

}

# for each beach ball:
# source: http://www.seismology.harvard.edu/CMTsearch.html
# table, created from ndk format
# fields: lat, lon, depth, magnitude, strike, dip, rake, unixtime
{
        BB_LON = $2;
        BB_LAT = $1;
        DEPTH  = ($3 / 6370);
        MAGNITUDE_FACT = ($4 - 5.5);     # typical: something like MAGNITUDE_FACT =
             magnitude - 6

        STRIKE = -$5;
        DIP    = -$6;
        RAKE   =  $7;

        NF = 3;

        # create gridpoints
        for( lat = 90; lat >= -90; lat-=STEPWIDTH )
        {
                for( lon = 0; lon < 360; lon+=STEPWIDTH )
                {
                        # location of each point = original location + rotation by its position on earth
                        # add STEPWIDTH/2 because colors are interpolated *between* points, i.e. it must lie exactly between two points
                        point_lon = lon + STEPWIDTH/2;
                        point_lat = lat;

                        # calculate cartesian coordinates
                        X = BB_RADIUS * MAGNITUDE_FACT * cos(point_lon * PI / 180) * cos(point_lat * PI / 180);
                        Y = BB_RADIUS * MAGNITUDE_FACT * sin(point_lon * PI / 180) * cos(point_lat * PI / 180);
                        Z = BB_RADIUS * MAGNITUDE_FACT * sin(point_lat * PI / 180);

                        # use correct position: rotate around x axis
                        X_ROT = X;
                        Y_ROT = Y * cos(CORR * PI / 180) - Z * sin(CORR * PI / 180);
                        Z_ROT = Y * sin(CORR * PI / 180) + Z * cos(CORR * PI / 180);

                        X = X_ROT; Y = Y_ROT; Z = Z_ROT;

                        # rotate: rake (rotate around x axis)
                        X_ROT = X;
                        Y_ROT = Y * cos(RAKE * PI / 180) - Z * sin(RAKE * PI / 180);
                        Z_ROT = Y * sin(RAKE * PI / 180) + Z * cos(RAKE * PI / 180);

                        X = X_ROT; Y = Y_ROT; Z = Z_ROT;

                        # rotate: dip (rotate around z axis)
                        X_ROT = X * cos(DIP * PI / 180) - Y * sin(DIP * PI / 180);
                        Y_ROT = X * sin(DIP * PI / 180) + Y * cos(DIP * PI / 180);
                        Z_ROT = Z;

                        X = X_ROT; Y = Y_ROT; Z = Z_ROT;

                        # rotate: strike (rotate around x axis)
                        X_ROT = X;
                        Y_ROT = Y * cos(STRIKE * PI / 180) - Z * sin(STRIKE * PI / 180);
                        Z_ROT = Y * sin(STRIKE * PI / 180) + Z * cos(STRIKE * PI / 180);

                        X = X_ROT; Y = Y_ROT; Z = Z_ROT;

                        # rotate according to latitude
                        X_ROT =  X * cos(-BB_LAT * PI / 180) + Z * sin(-BB_LAT *
                                        PI / 180);
                                Y_ROT =  Y;
                                Z_ROT = -X * sin(-BB_LAT * PI / 180) + Z * cos(-BB_LAT *
                                        PI / 180);

                                X = X_ROT; Y = Y_ROT; Z = Z_ROT;

                                # rotate according to longitude
                                X_ROT = X * cos(BB_LON * PI / 180) - Y * sin(BB_LON * PI /
                                         180);
                                Y_ROT = X * sin(BB_LON * PI / 180) + Y * cos(BB_LON * PI /
                                         180);
                                Z_ROT = Z;

                                X = X_ROT; Y = Y_ROT; Z = Z_ROT;

                                # move origin: add cartesian coordinates of earth
                                X += (EARTH_RADIUS - DEPTH) * cos(BB_LON * PI / 180) * cos
                                        (BB_LAT * PI / 180);
                                Y += (EARTH_RADIUS - DEPTH) * sin(BB_LON * PI / 180) * cos
                                        (BB_LAT * PI / 180);
                                Z += (EARTH_RADIUS - DEPTH) * sin(BB_LAT * PI / 180);

                                $1 = X;
                                $2 = Y;
                                $3 = Z;

                                print;
                        }
                }
}

# write the connectivity and the other remaining stuff
END {

# output polygons
print "POLYGONS " ((NRPOINTS - LONSTEPS) * NUMBER) " " (5 * (NRPOINTS - LONSTEPS)
     * NUMBER );

for( i = 0; i < NUMBER; i++ )
{

        for( lat = 0; lat < LATSTEPS - 1; lat++ )
        {
                for( lon = 0; lon < LONSTEPS - 1; lon++ )
                {
                        $1 = 4;
                        $2 = (lat * LONSTEPS + lon + i * NRPOINTS);
                        $3 = ( (lat + 1) * LONSTEPS + lon + i * NRPOINTS);
                        $4 = ( (lat + 1) * LONSTEPS + lon + 1 + i * NRPOINTS);
                        $5 = (lat * LONSTEPS + lon + 1 + i * NRPOINTS);

                        print;
                }

                $1 = 4;
                $2 = (lat * LONSTEPS + lon + i * NRPOINTS);
                $3 = ( (lat + 1) * LONSTEPS + lon + i * NRPOINTS);
                $4 = ( (lat + 1) * LONSTEPS + i * NRPOINTS);
                $5 = (lat * LONSTEPS + i * NRPOINTS);

                print;
        }

}

# output color data
print "POINT_DATA " (NRPOINTS * NUMBER);
print "SCALARS color float";
print "LOOKUP_TABLE colortbl";

for( i = 0; i < NUMBER; i++ )
{

        for( lat = 90; lat >= -90; lat-=STEPWIDTH )
        {
                for( lon = 0; lon < 360; lon+=STEPWIDTH )
                {
                        if( (1 == int( lon / 90)) || (3 == int( lon / 90)) )
                        {
                                print 1;
                        }
                        else
                        {
                                print 0;
                        }
                }
        }

}

# output color table (black and white)
print "LOOKUP_TABLE colortbl 2";
print "0.000000  0.000000  0.000000  1.000000";
print "1.000000  1.000000  1.000000  1.000000";

}
```



(a) Anisotropic case, isosurfaces at a deformation velocity of 10^{-9} ms^{-1}



(b) Anisotropic case



(c) PREM

Figure 6.4: Seismic wave propagation in a spherical section of the earth; these graphics show the velocity magnitude for a certain timestep.

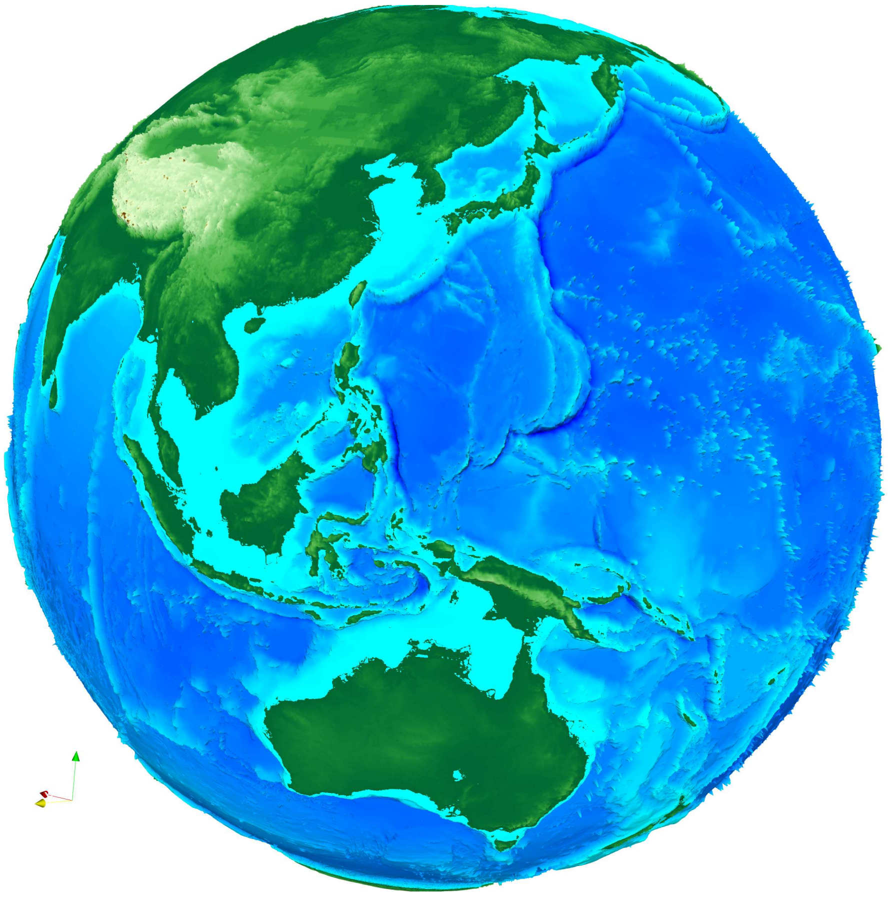


Figure 6.5: View of the earth, with heights 25 times exaggerated; note the deep sea trenches in the center and the sea mounts on the right.

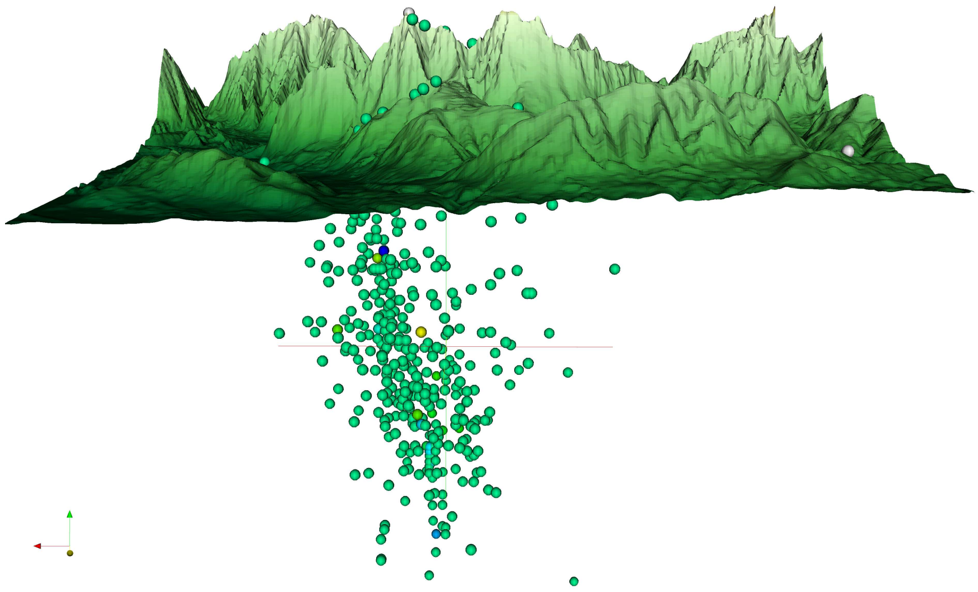
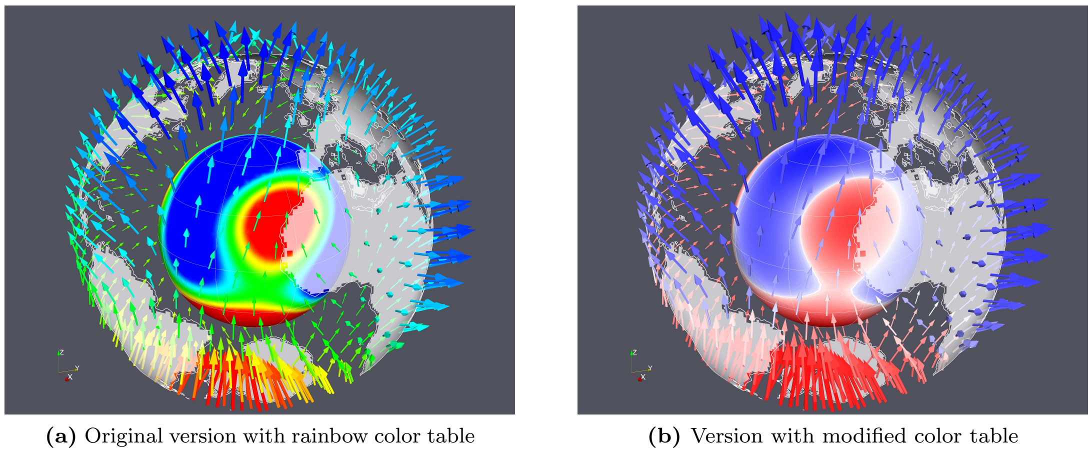


Figure 6.6: Hochstaufen, seen from the north, with the earthquake events of the year 2002 (colored balls) and seismometer stations (white balls). The heights are three times exaggerated.



(a) Original version with rainbow color table **(b)** Version with modified color table

Figure 6.7: Snapshot from the simulation of a magnetic field reversal; the original dipole field has been replaced by a multipole field, and the new components gain strength at the cost of the original field.

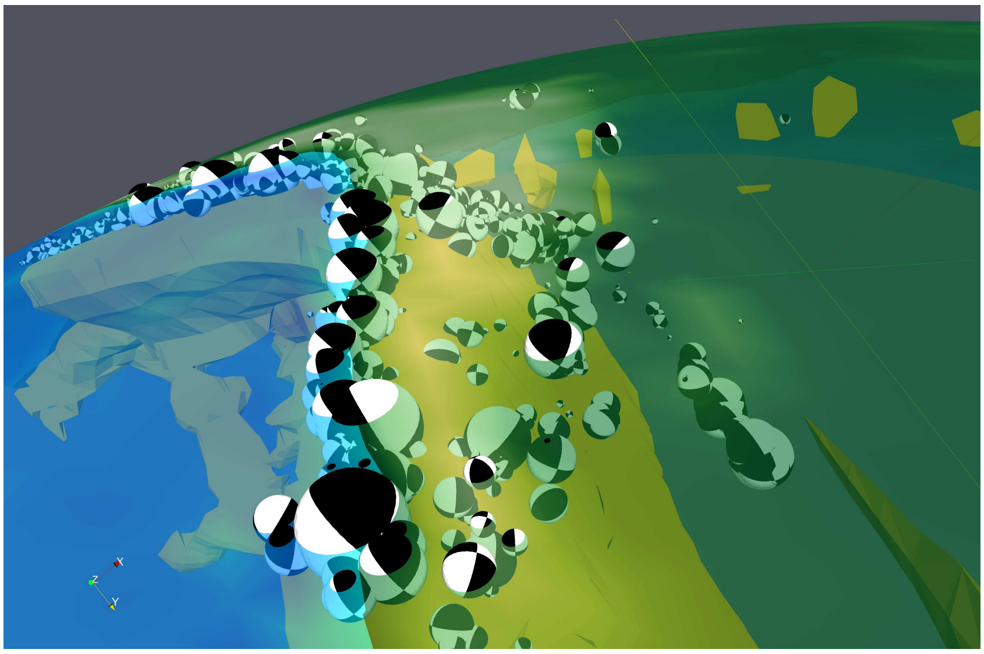


Figure 6.8: View from Chile to the north along the coast towards Peru; the map is displayed semi-transparent, below is to be seen the subducting slab and the beachballs indicating the hypocenters and their fault planes.

7. Conclusion and Outlook

Visualizing scientific data seems to be an easy task, at least at first glance. There is a plethora of software available which can be used to create graphics out of arbitrary data, which means there is an infinite number of degrees of freedom, yet also infinite degrees of failure.

So the question is not how to do it in general, but under certain — both restricting and challenging — boundary conditions. For example, there is the newly installed Geowall; so 3D visualizations are preferred, especially when they are interactive. But the available free Geowall software (esp. Immersaview) was not very convincing: The handling was cumbersome, and real interactivity was given only for very simple sceneries. Then someone asked: "Can you also cut through the objects?" This was the point. Of course, I can. But there's nothing below; VRML deals only with surfaces. But these have to be generated before — I had simply taken them for granted.

However, there are indeed programs that *can* process volumetric data in realtime and interactively, notably Paraview. But for a long time, it was uncertain to me if this program can be of any use for a Geowall; there were only vague rumors about a patch enabling a Geowall-compatible stereo mode, not more. It turned out that Paraview and the underlying VTK are very powerful and flexible, that they can be adapted to the Geowall, and using MPI and Paraview's client-server mode, they can handle datasets that are several magnitudes bigger than everything that can be done with VRML. It could have been so easy — but these programs do not run out of the box when used with these advanced features. Although there are books available for Paraview and VTK, the documentation is still poor, compared to the possibilities they offer. Time-consuming investigation was necessary much too often. A quite steep learning curve in an area with so many different requirements and few simple recipes.

What comes next? In our hardware setup, Paraview was not finished when loaded with nearly 100 million grid points (which is about three magnitudes more than what's possible with VRML). And if, the cluster could be extended with more machines and the clients could be equipped with more RAM and maybe better graphics cards (which could even be coupled in pairs) — if rendering with multiple clients or server-side rendering works, there seems to be no further bottleneck in range. Also the software has still much going for it. The small VTK scripts presented here are just the beginning; a lot of automation is possible. And the next major release of Paraview, promising many new features, is in sight, too. The tools — once running — are good and still improving.

Whereas the steps from the data file to the screen become easier with the help of high-level toolkits, the gap between raw data (i.e. what the simulation programs produce) and the input for visualization software still exists. A kind of "glue software" would be helpful. I was quite impressed by the software package Holodraw [Simkin, 2006] which offers a very elegant way to convert plain ASCII data files into the much more complicated VRML format. A similar high-level preprocessing and scripting extension for the (much more complicated) VTK package could be very useful.

Anything else? Yes. As mentioned, simple recipes are not sufficient for creating visualizations, the problems are too different. A lot of experience is necessary in this broad topic, and thus much more cooperation for sharing it.

A. Bibliography

AHRENS, J. ; LAW, C. ; SCHROEDER, W. ; MARTIN, K. ; PAPKA, M.: *A Parallel Approach for Efficiently Visualizing Extremely Large, Time-Varying Datasets.* `http://citeseer.ist.psu.edu/ahrens00parallel.html`. Version: 2000

BAUMGARDNER, John R. ; FREDRICKSON, P. O.: Icosahedral discretization of the two sphere. In: *Siam J. Numer Anal.* (1985)

BIDDISCOMBE, John: *vtkCSCNetCDF.* `http://www.cscs.ch/a-display.php?id=1168`. Version: 2006

BOURKE, Paul: *Portable Rear Projection Stereoscopic Display: Stereographics Theory.* `http://local.wasp.uwa.edu.au/~pbourke/stereographics/vpac/theory.html`. Version: 2002

BUNGE, Hans-Peter ; BAUMGARDNER, John R.: Mantle convection modeling on parallel virtual machines. In: *Computers in Physics* (1995)

BUNGE, Hans-Peter ; RICHARDS, M. A. ; BAUMGARDNER, John R.: A sensitivity study of 3-D spherical mantle convection at 10exp8 Rayleigh number: Effects of depth dependent viscosity, heating mode and an endothermic phase change. In: *J. Geophys. Res.* (1997)

BUNGE, Hans-Peter ; RICHARDS, M. A. ; BAUMGARDNER, John R.: Mantle circulation models with sequential data-assimilation: Inferring present-day mantle structure from plate motion histories. In: *Phil. Trans. Roy. Soc. A* (2002)

BUNGE, Hans-Peter ; RICHARDS, Mark A. ; BAUMGARDNER, John R.: The effect of depth dependent viscosity on the planform of mantle convection. In: *Nature* (1996)

CEDILNIK, Andrej ; GEVECI, Berk ; MORELAND, Kenneth ; AHRENS, James ; FAVRE, Jean: Remote Large Data Visualization in the ParaView Framework. In: HEIRICH, A. (Hrsg.) ; RAFFIN, B. (Hrsg.) ; SANTOS, L. P. (Hrsg.): *Eurographics Parallel Graphics and Visualization 2006*, 2006, S. 162–170

FLANAGIN, Maik: *vtkGeography.* `http://www.cs.uno.edu/~maik/vtkGeography/`. Version: 2005

GEUZAINE, Christophe: *GL2PS.* `http://www.geuz.org/gl2ps/`. Version: 2006

GHOSH, Satrajit: *vtkAutomex.* `http://speechlab.bu.edu/VTK.php`. Version: 2004

GRASS GIS 6.1 Reference Manual. `http://grass.itc.it/grass61/manuals/html61_user/v.out.vtk.html`. Version: 2006

Gropp, William ; Lusk, Ewing ; Ashton, David ; Buntinas, Darius ; Butler, Ralph ; Chan, Anthony ; Ross, Rob ; Thakur, Rajeev ; Toonen, Brian: *MPICH2 User's Guide*. http://www-unix.mcs.anl.gov/mpi/mpich/downloads/mpich2-doc-user.pdf

CMT Catalog Search. http://www.seismology.harvard.edu/CMTsearch.html. Version: 2006

Igel, Heiner ; Nissen-Meyer, Tarje ; Jahnke, Gunnar: Wave propagation in 3D spherical sections: effects of subduction zones. In: *Phys. Earth Planet. Int.* (2002)

VTK File Formats. http://www.vtk.org/pdf/file-formats.pdf. Version: 2006

The VTK User's Guide, Version 4.4. Kitware Inc., 2004. – ISBN 1930934130

Kraft, Toni ; Wassermann, Joachim ; Igel, Heiner: High-precision relocation and focal mechanism of the 2002 rain-triggered earthquake swarms at Mt. Hochstaufen, SE-Germany. In: *Geophys. J. Int.* (2006). – in press

Kraft, Toni ; Wassermann, Joachim ; Schmedes, Eberhard ; Igel, Heiner: Meteorological triggering of earthquake swarms at Mt. Hochstaufen, SE-Germany. In: *Tectonophysics* (2006). http://www.sciencedirect.com/science/article/B6V72-4K9C55N-1/2/56a918df3aec06670e36cae850a0c777

Lorensen, Bill: *vtkRIBExporter*. http://www.crd.ge.com/~lorensen/vtkrib/. Version: 2000

Octaviz. http://octaviz.sourceforge.net/

Rajlich, Paul: *vtkActorToPF*. http://brighton.ncsa.uiuc.edu/%7Eprajlich/vtkActorToPF/. Version: 2006

Schroeder, William J. ; Martin, Kenneth M. ; Lorensen, William E.: The Design and Implementation of an Object-Oriented Toolkit for 3D Graphics and Visualization. In: Yagel, Roni (Hrsg.) ; Nielson, Gregory M. (Hrsg.): *IEEE Visualization '96*, 1996, 93–100

Simkin, Marvin: *Draw Your Data in 3D with HoloDraw*. http://www.holodraw.org/. Version: 2006

Sites, Chuck: *vtkFVR*. http://mecca.louisville.edu/Staff/Sites/vtkFVR.html. Version: 2003

Steinwand, Daniel ; Davis, Brian ; Weeks, Nathan: GeoWall: Investigations into Low-cost Stereo Display Systems. In: *USGS Open File Report* (2002). http://geowall.geo.lsa.umich.edu/papers/Geowall.pdf

Stockwell, John W. ; Cohen, Jack K.: *The New Seismic Unix User's Manual*. ftp://ftp.cwp.mines.edu/pub/cwpcodes/sumanual_600dpi_a4.pdf. Version: 2002

Szczerba, Dominik: *VTKMatlab Mini HOWTO*. http://www.vtk.org/Wiki/VTK/VTKMatlab. Version: 2006

B. Instructions for the Example Datasets

As already mentioned, it is not feasible to write a general recipe how to convert arbitrary datasets into three-dimensional VTK models. But I have provided several different examples together with all datasets and tools that have been used; they may serve as an assistance for processing similar input data or as an inspiration how to deal with both ASCII and binary input data, how to write the cell connectivity and so on. All mentioned examples can be found on the attached DVDs (in the directory *Examples*) as well as in the computer network at the LMU geophysics section in the directory */import/holodeck-data/3D-Data/*.

In every example directory there is a subdirectory *Source* which contains the original dataset, and a subdirectory *Tools* which contains all scripts that have been used to convert the data to a VTK file format. Here is a short description of the datasets and the tools that have been used create the Paraview files.

B.1. Wave Propagation

Directory: *Andreas_Fichtner* (with two subdirectories containing two simulations)

Source Data:

- Point coordinates of the grid axes (*xco0–xco17* for the x axis, *yco0–yco17* for y axis and *zco0–zco17* for z axis); units: polar coordinates in radians (x and y axis), meters (z axis).
- Data files: *v_phi_block**, *v_theta_block** and *v_r_block**; units: deformation velocities in m/s.

Conversion Steps:

- *make_coordinates.sh*: This script generates all combinations of coordinates and writes them into files with the names *xyz_0.dat–xyz_17.dat*.
- *xyz2vtkunstructured.sh*: This script converts the polar coordinates to cartesian coordinates and writes them into ASCII VTK files; the command line parameter indicates the block number; e.g. “12” means that the file *xyz_12.dat* should be converted.
- *calculate_vvect.sh*: This script takes the VTK file with the given number and appends the data values – as vectors, as scalar values and the single components; additionally, the file is converted to the binary VTU format using *convert2vtu.tcl*.

B.2. Global Earthquake Distribution

Directory: *Earthquakes_Global*

Source Data:

- *jan76_dec05.ndk*: A file with all earthquakes from January 1976 until December 2005 (from `http://www.globalcmt.org/CMTfiles.html`).

Conversion Steps:

- *ndk2table.awk*: This script takes a *.ndk file and reformats it to a table with all relevant fields (location, time, magnitude, source mechanism).
- *make_beachballs_from_file.awk*: This script takes a table (generated e.g. with `ndk2table.awk`) and creates a beachball for each earthquake event in a VTK file. First, all points of a beachball sphere in cartesian coordinates are calculated; then, the beachball is rotated (strike, dip, rake) and shifted to its final position on the earth's surface.
- *create_time_series.sh*: This script takes a table of earthquakes and calls in a loop *make_beachballs_from_file.awk*, each time with an increasing number of earthquakes from the list (starting with the uppermost). The result is a set of VTK files which animate the emergence of the earthquakes.

B.3. ETOPO2

Directory: *ETOPO2*

Source Data:

- *swg_9432.xyz*: This file has three columns.
  1. latitude (in degrees), in steps of two arc minutes
  2. longitude (in degrees), in steps of two arc minutes
  3. height (in m above sea level)

  A problem is that some values appear twice since the longitude runs from $-180°$ to $+180°$. This has to be considered during processing, if the source data should be used without modifications.

Conversion Steps:

- *extract_pieces.awk*: Because the input data is too large for one single VTK file, this script is used to split the ETOPO2 dataset into several pieces; the input is read from stdin and the contiguous pieces are written into several output files (*etopo_xxx.xyz*). By default, 162 pieces are created (latitudes are divided into 9 sections, longitues are divided into 18 sections).

- *xyz2vtkpoly.sh*: This script takes one of the generated pieces (**.xyz*) (filename given on the command line) and converts it into the vtk format (ASCII, polydata) (output on stdout).
- *convert2binary.tcl*, *convert2vtp.tcl*: With these scripts, the ASCII VTK files can be converted into the binary VTK format (=> smaller files) or XML VTP format.
- *separate_land_sea.tcl*: This script takes one VTU file and extracts the parts above and below sea level. The input file name is the first command line parameter, the file name for the land file is the second parameter, and the file name for the sea file is the third parameter.
- *make_land_sea.sh*: A wrapper script which calls the previous script for all available VTP files, creates the PVTP files and changes the variable name in the sea files from "*altitude*" to "*sea_altitude*".
- *triangulate_decimate.tcl*: This script takes one VTP file and reduces the number of polygons. It applies the triangulate filter (trapezoids to triangles), the decimate filter (reduces the number of polygons) and the stripper filter (triangles to triangle strips).

B.4. ETOPO5

Directory: *Topo-new*

Source Data:

- *topo_5min.xyz*: ETOPO dataset, created with GMT's *grd2xyz*

Conversion Steps:

- *downsample-xyz.sh*: This script downsamples a **.xyz* file by omitting grid points in both latitude and longitude direction.
- *xyz2vtkpoly.sh*: This script takes a **.xyz* file and converts it to a VTK polydata file (converts polar to cartesian coordinates, calculates the connectivity).

B.5. Hochstaufen

Directory: *Hochstaufen*

Source Data:

- *RH_topo_ev.xyz*: ASCII file with a digital elevation model of the area around Hochstaufen; 1st column = UTM easting, 2nd column = UTM northing, 3rd column = height above sea level.

- *bay_gk.dat*: list of the seismometer stations around Hochstaufen; 1st column = UTM easting, 2nd column = UTM northing, 3rd column = height above sea level, 4th column = station short name
- *BULL2002_NLLCC_all.dat*: earthquake events in the year 2002; the meaning of the columns is described in the header

Conversion Steps:

- *xyz2vtkpoly.sh*: This script creates a VTK (ASCII) file from the grid data (UTM coordinates) in the file given as the first command line parameter; the output is written to stdout. The height exaggeration is set to 3.
- *create_glyphs.tcl*: This script reads a VTK file (given by the first command line parameter) and creates a sphere at every grid point. The resulting VTK file (name: second command line parameter) is then saved.
- *hochstaufen_make_all.sh*: This script generates all VTK files. First, *xyz2vtkpoly.sh* is called to create a VTK file of the landscape. Then, the station coordinates are written into a VTK file and processed with *create_glyphs.tcl*; the result is a VTK file with spheres indicating the seismometer stations. Third, the earthquakes are processed. In a loop, all the events (except those which could not be well located) before a certain time are written into a VTK file and processed with *create_glyphs.tcl*; the result is an animatable series of files where the spheres that indicate the earthquakes appear accordingly to their occurrence.

B.6. Magnetic Field Reversal

Directory: *Magnetic_Field_Reversal*

Source Data:

- The provided data are already VTK files, but with the default color table.

Conversion Steps:

- *change_colortable.awk*: This script normalizes the data values in a VTK file; the minimum and maximum values are conveyed via environment variables.
- *create_lookup_table_france.awk*: This script creates a VTK color table with a nonlinear transition from blue via white to red.
- *convert2binary.tcl*: This script converts an ASCII VTK file into the binary VTK format.
- *create_glyphs.tcl*: This script creates vector arrows according to the vector value at every grid point of the input file. The filenames are given as command line parameters.

- *convert_cmb_files.sh*: This script reads all *radcomp_⋆.vtk* files, determines the minimum and maximum values of the field strength, and normalizes the data values using *change_colortable.awk*. After that, a color table (created with *create_lookup_table_france.awk*) is appended to the VTK file, and the file is converted to binary (*convert2binary.tcl*).
- *convert_surface_files.sh*: This script is similar to *convert_cmb_files.sh*, but since the surface files have additionally a vector field (magnetic field vectors), some additional processing steps are necessary. First, the VTK file is converted to a set of vector arrows using *create_glyphs.tcl*; after some modifications (done with *sed*), a new color table is appended also to this file (created with *create_lookup_table_france.awk*), and the resulting file is converted to the binary format (with *convert2binary.tcl*).

B.7. Plate Boundaries

Directory: *Plate_Boundaries*

Source Data:

- *Plate_Boundary_Polygons_org.dat*: GMT file with the plate boundary polygons (see ftp://element.ess.ucla.edu/PB2002/)

Conversion Steps:

- *gmtpolygon2vtk.awk*: This script reads the GMT file, counts the number of polygons, converts the polar coordinates into cartesian coordinates, and writes them into a VTK file together with their connectivity.
- *gmtpolygon2vtk_deep.awk*: Similar to *gmtpolygon2vtk.awk*, but each point is copied to a deeper layer in the earth, so the final result is not a polygonal line, but a polygon made up of the two lines.

B.8. San Francisco Faults

Directory: *San_Francisco_Faults*

Source Data:

- Files of the faults around San Francisco in the tsurf format (*⋆.ts_deg83* and *⋆.ts_km83*).

Conversion Steps:

- *tsurf_deg2vtk.awk*: This script takes **.ts_deg83* files and converts them into VTK polydata files. Since the file concepts are similar, only the number of points and polygons must be counted and the polar coordinates must be converted into cartesian coordinates.

B.9. Seismic Profile

Directory: *Seismic_Profile*

Source Data:

- *mod00.3d*: This binary file contains the raw seismic velocities of a structured grid.

Conversion Steps:

- *3d2vtk.sh*: This script writes the VTK header; the binary data is appended using the tool swapbytes from the Seismic Unix package. Because a left-handed coordinate system is used, the resulting file must be mirrored afterwards (can be done with Paraview).

B.10. South America Slab Earthquakes

Directory: *South_America_Slab_Earthquakes*

Source Data:

- *jan76_dec05.ndk*: A file with all earthquakes from January 1976 until December 2005 (from `http://www.globalcmt.org/CMTfiles.html`).
- map tiles from ETOPO2 (*etopo_*_land.vtp*, *south_america.pvtp*)

Conversion Steps:

- *ndk2table.awk*: This script takes a **.ndk* file and reformats it to a table with all relevant fields (location, time, magnitude, source mechanism). Only earthquakes with a magnitude greater than 5.0 and a location around South America are processed.
- *make_beachballs_from_file.awk*: This script takes a table (generated e.g. with *ndk2table.awk*) and creates a beachball for each earthquake event in a VTK file. First, all points of a beachball sphere in cartesian coordinates are calculated; then, the beachball is rotated (strike, dip, rake) and shifted to its final position on the earth's surface.

B.11. Tecplot

Directory: *Tecplot_Josep*

Source Data:

- *Mesaverde_noQ_O3-000001005.dat*: file in Tecplot format (see ftp://ftp.tecplot.com/pub/doc/tecplot/360/dataformat.pdf)

Conversion Steps:

- *tecplot-FE3D2vtk.awk*: This script converts a tecplot file to a VTK unstructured grid file. Because the point coordinates are saved together with the associtated data values in this file format, the data values are written to a temporary file and appended to the VTK file after the output of the point coordinates and cells is finished.

C. Contents of the Attached DVDs

/

`Documentation`	technical documentation and mentioned scientific papers
`Examples`	the mentioned example visualizations including source data and processing scripts
`MPI-tools`	useful shell scripts for parallel processing
`Software-32bit`	Tarballs of the mentioned software: • Mesa (normal and mangled), compiled as static libraries • mpich with patched message size • Paraview with stereo support (clone mode) and modified eye angle, as a normal version, with MPI and with MPI and Mesa • VTK
`Software-64bit`	Tarballs of 64-bit software: • Mesa, compiled as static libraries • mpich with patched message size • Paraview with MPI and Mesa
`TERRA`	TERRA code with all mentioned modifications for VTK output
`VTK-demofiles`	a simple VTK unstructured grid file in different file formats

D. Acknowledgments

I would like to thank . . .

- *Heiner Igel*, for assigning this thesis to me and for allowing a lot of freedom to find my own way.
- *Hans-Peter Bunge*, for providing the workplace, the data of his newest simulations and a lot of patient help with the code of TERRA. Also his interest in my work and some very good conversations have given me motivation that I have sometimes needed.
- *Bernhard Schuberth*, for some very pleasant and fruitful cooperations. I would not have dared to tamper with Fortran without his assistance and encouragement, and quite some times when the bugs seemed unsolvable, we could finally handle them together.
- *Jens Oeser*, for the fantastic computing infrastructure that he has built up. I remember quite well the horrible conditions of the computers before he has renewed the Linux installations — nothing really worked, the software was outdated and broken. Also whenever I had any problem or question, Jens knew an answer and gave me a hint or solved it instantaeously. Without him, I would have had to fight not only the problems in my software, but also in the rest of the systems.
- *Markus Treml*, for sharing a lot of experience during his Ph. D. thesis. He is one of the those rare people who tell you the things you will have to know in the future, without having asked for them — and even without knowing about them. With some thinking, it is easy to go the next steps. But it needs the advice of the experienced to lern about alternative ways and to discover the shortcuts.
- *Andreas Fichtner*, *Roman Leonhardt* and *Toni Kraft* for providing their datasets of wave propagation and magnetic reversal simulations and measurements of swarm earthquake hypocenters, respectively.
- My friends and flat mates *Michael Wack*, *Basti Mühlbauer* and *Kerstin Reimer* who made the life at home easy and very pleasant. Things could have been much more annoying without mutual assistance, and I do appreciate that.

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Hilfsmittel und Quellen angefertigt habe.

München, den